De la conception objet au langage C++

Marc Zonzon

Année 2003-2004

Rev: 1.02

Table des matières

Avant-propos

Ce polycopié tente d'introduire à la fois le langage **C++** et une discipline de programmation qui permette de dériver d'une spécification objet un programme de qualité.

Avant de faire notre premier pas de programmeur nous devons faire le point.

Quelle est la base de notre travail ? Quel est le but ? Quel est le chemin ? Quelles sont nos forces ? Ces questions doivent être examinées avant toute action.

Nos ordinateurs sont de beaux outils, mais sans l'esprit ils seraient dépourvus de la moindre utilité, sans l'esprit, ils ne pourraient même pas exister.

La charrue est un bel outil, mais sans terre, semence, eau et soleil, la charrue seule ne produit pas de récolte. Seule l'intention, et l'action du laboureur peut en faire une arme qui détruit la fertilité du sol, un objet de collection dans une vitrine, ou un merveilleux remède à la faim.

Pour évaluer la récolte à venir avant d'entreprendre la culture, nous devons tout d'abord considérer les objectifs les plus lointains et les plus larges avant ceux plus proches et plus étroits qui leur sont subordonnés.

À quoi sert un composant parfait s'il ne s'intègre pas au produit ? À quoi sert un produit parfait s'il ne correspond pas à l'usage qui en est fait ? À quoi sert son utilisation si elle produit peine et douleur ?

L'évaluation du but est difficile du fait de notre myopie, qui nous fait nous arrêter à ce qui est près de nous et nous semble agréable et aisé à saisir. Nous sommes souvent comme cet homme qui, ayant perdu ses clés de nuit au milieu de la rue, va les chercher sous le réverbère parce que c'est l'endroit où la recherche est la plus aisée.

Aussi, quand nous dirigeons la lumière de la connaissance par la lentille de la science et le prisme de la technique, nous devons nous garder de ces lumières rasantes qui éclairent un détail mais jettent l'horizon dans l'ombre.

Il nous aidera de nous rappeler cet avertissement de Rabelais : « Science sans conscience n'est que ruine de l'âme.» Cela évitera que le *Génie logiciel* ne se change en mauvais génie.

Le génie logiciel nous impose de travailler en groupe car l'objectif est trop complexe pour être atteint par un seul individu en un temps raisonnable. Pour tout

travail partagé, des motivations divergentes peuvent ruiner nos efforts ; mais si les objectifs sont vastes, si les buts sont clairs, nous pourrons nous convaincre de leur validité et il pourra s'établir une coopération fructueuse.

Pour travailler ensemble il nous faut communiquer. Et parce qu'il demande de faire coopérer tout un groupe d'individus, le génie logiciel est avant tout une discipline de communication.

La spécification de l'application sert de base à l'analyse, l'analyse doit guider la conception, la conception se traduit en programmation, la programmation est relue par une vérification et explorée par des tests, le produit est employé par des utilisateurs, les erreurs reportées au service de maintenance et ainsi de suite. Le *cycle de vie* du logiciel est là, dans ce bourdonnement continuel de la communication. Le logiciel lui-même, en perpétuelle modification n'a pas d'autre existence que son processus de développement.

Pour qu'il y ait communication il nous faut un langage commun. Mettre au point de tels langages, des normes de communication, faciliter leur emploi, vérifier leur correction, constitue l'aspect technique essentiel du génie logiciel.

Le programmeur qui écrit ses instructions ne parle pas seulement à un compilateur stupide, mais il parle à ses collègues qui vont utiliser ce module, le vérifier, le tester, l'intégrer dans leur système, à celui qui demain ou dans vingt ans modifiera le code, à l'utilisateur qui va employer le programme, à tous ceux qui en éprouveront les effets.

Nous devons dans notre travail nous poser sans cesse les questions : «comment serai-je compris ?», «Quels seront les effets de ce que je produis ?».

Pour aider celui qui tentera de nous déchiffrer et pour délivrer un produit fiable il est indispensable d'adopter des normes. Elles forment ce *code social* qui nous permet de nous comprendre, et elles contiennent le savoir-faire accumulé par de nombreux programmeurs.

Dans cette nouvelle version de mon cours j'ai incorporé et expliqué les normes recueillies par Mats Henricson et Erik Nyquist[1].

Ces conseils forment la discipline, qui est le rempart qui protège notre travail : dans ce cadre nous pouvons polir la lentille de l'intelligence, mais ni la pierre ni le verre ne peuvent nous guider, seule la lumière a cette capacité.

Le génie logiciel est un domaine difficile et il vous faudra probablement longtemps pour l'apprivoiser.

Souvent, alors que nous croyons enfin arriver au but, les spécifications, le matériel, l'équipe ou l'entreprise ont changé et il nous faut recommencer. Pour réaliser un produit de qualité il nous faut accepter cette instabilité, les seuls produits finis sont ceux qui ne servent plus.

Le laboureur, qui sait que l'été sera suivi de l'hiver, chaque printemps retourne

[1] *Industrial strength C++* Prentice Hall 1997

son champ. Notre travail sera plus aisé et plus utile si nous savons que, comme nous-mêmes, ce que nous produisons, n'apparaît que pour une courte saison ; et que comme le laboureur nous travaillons dans le double but de nourrir notre large famille et de laisser à ceux qui viendront demain une terre fertile.

Ce souhait, je le fais ici pour mon propre travail de laboureur-enseignant.

Mode d'emploi.

Conseils pour l'apprentissage de C++ .

Apprenez **C++** par étape. Commencez par de petits programmes, mais n'hésitez pas à utiliser dès le début des concepts de haut niveau et àutiliser la bibliothèque standard.

Ne faites pas de la programmation assembleur en **C++** , il n'est pas nécessaire de connaître **C** pour apprendre **C++** ; mais si vous le connaissez, réservez les manipulations directes de la mémoire pour la programmation système de bas niveau en **C** ; en **C++** les couches abstraites permettent la même efficacité, mais sans sacrifier fiabilité et productivité.

Ne laissez au préprocesseur que la gestion de l'inclusion des en-têtes, mais, même si vous êtes un programmeur **C**, ne définissez ni constantes ni fonctions par des macro-instructions : **C++** a les outils appropriés pour cela. Réservez à plus tard les pointeurs, la mémoire libre : ils sont traités dans ce texte, mais ne les étudiez que quand vous aurez compris les problèmes de construction/destruction et de sécurité des exceptions, etla manière dont la bibliothèque standard résout ces problèmes.

Les références et les conteneurs de la bibliothèque standard vous permettent dans la grande majorité des cas de ne pas utiliser directement les pointeurs et l'allocation mémoire. Oubliez en particulier les `char*` utilisez des `string` (mais bien sûr, vous pouvez employer des littéraux `const char*` !)

En revanche apprenez dès le début comment déclarer et initialiser les variables, comment se font constructions et destructions. Utilisez tout de suite les espaces de noms.

Apprenez l'utilisation des patrons dès le début car la bibliothèque standard les utilise intensivement ; mais laissez la conception de nouveaux patrons pour le moment où vous aurez maîtrisé leur emploi.

Utilisez tout de suite les exception : les erreurs n'attendront pas que vous soyez expert pour se produire ; tous les incidents ne peuvent être traités par l'idiome du «ctrl-alt-suppr».

Vous devez dès le début apprendre à tester vos programmes : le test devrait

être prêt avant, ou au plus tard en même temps que le programme, il vous aidera à produire des programmes fiables. Cependant il faut savoir que même les meilleurs tests n'attraperont qu'une partie des erreurs. Le débogage même, est souvent impossible si vos programmes ont des contraintes temps réels, directes ou indirectes comme celles qu'induit l'utilisation des *threads*.

La meilleure vérification est celle du code source ; le meilleure preuve est celle du programme ; une démonstration est au mieux un argument commercial.

Fournissez systématiquement pré-conditions post-conditions et invariants : si votre programme doit être réparé, modifié ou utilisé par d'autres modules ils seront indispensables.

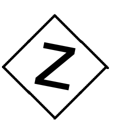
Mieux vaut se débarrasser d'un module de conception défectueuse ou non documenté.

Pour la plupart nous apprennenont lentement : des milliers d'heures de programmation sont souvent nécessaires pour devenir expert. On apprend beaucoup à étudier de bons exemples comme on stagne à ne suivre que des habitudes, que cela soit les siennes ou celles des autres. Enfin on perd son temps à dénigrer d'autres méthodes de conceptions, d'autres langages : mieux vaut retenir le meilleur de chacun.

Structure de ce manuel

Le premier chapitre est conçu comme une introduction rapide, un apéritif pour le débutant, il doit lui permettre de traiter rapidement des problèmes non triviaux. Beaucoup de notions sont abordées, aucune n'est traitée en détail.

La suite est une exploration systématique des concepts. Elle n'est pas conçue pour une lecture linéaire de la première à la dernière page, mais devra être explorée par couches successives.



Certaines parties sont marquées par un sigle comme celui qui figure en marge ci-contre pour indiquer qu'il s'agit de considérations plus difficiles, qui ne sont pas appropriées à une première lecture.

Ce texte a été conçu pour accompagner un cours qui doit guider l'éudiant, et pour servir ultérieurement de référence. Il n'est sûrement pas adapté à l'auto-apprentissage du **C**++ , sauf pour qui a déjà une expérience d'un langage analogue.

Les deux derniers chapitres sont une présentation incomplète, mais cependant assez étendue de la bibliothèque standard, qui devrait faciliter son utilisation. La table des matières et un index permettent de retouver rapidement les concepts cherchés. (Dans la version *pdf* ce sont des liens hypertexte.)

Chapitre 1

Introduction à la bibliothèque standard et aux classes

Dans ce chapitre nous allons donner quelques exemples d'utilisation de la bibliothèque standard. Le programmeur d'application utilise habituellement des composants pré-assemblés, et les mécanismes du langage permettent d'assembler ces composants.

Pour résoudre un problème réel, même s'il est de dimension modeste, il serait long et peu sûr de le faire avec seulement les briques de base que sont les types prédéfinis, cela serait comme construire un pont en utilisant des planches et des clous.

Bien entendu pour fabriquer les composants les plus simples, il faut maîtriser l'emploi des pointeurs, des tableaux, de l'allocation en mémoire libre, et même parfois des *unions* et *fields*, mais ce n'est pas ce que nous aborderons d'abord, car cela n'est pas le but premier de la programmation objet ; de plus ces outils ont souvent été étudiés avec le langage **C** qui les utilise d'une manière semblable à **C**++ .

L'utilisation de *classes* pourrait simplifier et rendre plus sûrs les programmes de ce chapitre d'introduction, si nous avons sacrifié l'efficacité et l'élégance pour permettre une lecture plus aisée par le programmeur débutant en **C**++ nous en envisagerons l'emploi en fin de chapitre.

1.1 Les chaînes de caractères et les entrées sorties

1.1.1 Lecture et écriture d'une chaîne de caractères

Notre premier exemple (Programme 1.1) va nous montrer comment lire et écrire une chaîne de caractères en C++.

Programme 1.1 Lecture et écriture du nom.

```
1 #include <string>
2 #include <iostream>
3 int main(){
4   using namespace std;
5   string nom;
6   cout << "quel est votre nom"<<endl;
7   cin >> nom;
8   cout << "Bonjour "<<nom<<endl;
9   return 0;
10 }
```

Pour utiliser les chaînes de caractères de la bibliothèque standard nous devons commencer par inclure l'en-tête `<string>` (ligne 1) ; pour les entrées sorties il s'agit de `<iostream>` (ligne 2).

Notre programme principal est simplement une fonction de nom `main` et de type de retour entier. La valeur de retour (ligne 9) est retournée au *shell*, une valeur de zéro indique qu'il n'y a pas d'erreur.

Les données sont lues sur le flot d'entrée standard `cin` (c'est-à-dire au clavier) par l'opérateur >> (ligne 7) et ils sont écrits sur le flot de sortie `cout` (c'est à dire l'écran) par l'opérateur << (ligne 6) .

Le spécificateur `endl` (ligne 6) indique qu'il faut passer à la ligne dans le flot de sortie.

Les chaînes de caractères qui sont incluses telles quelles dans le programme, que nous nommerons littéraux chaînes de caractères, sont constituées de suites de caractères délimitées par des `"` (lignes 6, 7).

L'opérateur >> appliqué à une variable chaîne de caractère (ligne 7) la remplit avec les caractères entrés jusqu'au premier espace ou fin de ligne.

Les flux d'entrée-sortie `cin` et `cout` et les opérateurs appartiennent à la librairie standard (`std`) et leur nom complet est `std::cin` et `std::cout`, mais la ligne 4 importe les noms de l'espace `std` et permet de les utiliser sans préfixe à l'intérieur de la fonction `main`.

Nous ne pouvons pas donner notre premier et second prénom à ce programme, ni notre prénom et nom de famille, puisqu'il s'arrête au premier espace. Pour lui faire lire une ligne entière nous devons remplacer la ligne ligne 7 par :

```
7   getline(cin, str);
```

1.1.2 Lecture d'un type prédéfini.

Programme 1.2 Lecture écriture de l'âge

```
1 const int age_max(125);
2
3 int main(){
4   int age;
5   int retour(0);
6   std::cout << "Quel est votre age ?"<<std::endl;
7   std::cin >> age;
8   std::cout << age << " an";
9   if(age >=0 && age < age_max){
10    if (age >1) {
11      std::cout <<'s';
12    }
13    std::cout << " est un bien bel âge." <<std::endl;
14  }else{
15    std::cout << " cela n'est pas possible" <<std::endl;
16    retour=1;
17  }
18  return retour;
19 }
```

La bibliothèque d'entrée-sortie peut, bien entendu, lire et écrire, outre les chaînes, tous les types prédéfinis, par exemple les entiers comme l'illustre le programme 1.2.

Ce petit programme illustre aussi la syntaxe d'initialisation d'une variable (ligne 5) et l'emploi des constantes (ligne 1).

1.1.3 Quelques opérations sur les chaînes de caractères.

Les `string` de la bibliothèque standard ont de nombreuses opérations prédéfinies (§11.3 et §11.4), les plus utilisées sont les comparaisons que nous pouvons simplement effectuer avec les opérateurs usuels (`==`,`<`, `<=`, ...). Nous utiliserons souvent aussi la concaténation : on concatène une chaine à la fin d'une variable chaîne avec l'opérateur `+=`, et deux chaînes par l'opérateur `+`.

Le programme §1.3 donne aussi l'utilisation de la recherche d'une sous-chaîne et de son remplacement.

Programme 1.3 Opérations sur les chaînes

```
#include <string>
#include <iostream>
std::string prenom("Gottlieb");
std::string nom("Mozart");
int main(){
   using std::cout; using std::cin; using std::endl;
   nom=prenom+' '+nom;
   cout << nom <<endl;
   cout<<nom.substr(0,4)<<endl;
   int i(nom.find('M')-1);
   nom.replace(0,i,"Amadeus");
   cout << nom <<endl;
   return 0;
}
```

Programme 1.4 interface de la feuille de notes.

```
// $Id: chap-intro.tex,v 1.2 2002/09/06 13:40:38 marc Exp $
#ifndef FEUILLENOTES_HH
#define FEUILLENOTES_HH
void consulterNote();
void entrerNote();
void listerNotes();
#endif
```

1.2 Les vecteurs

Dans cette section nous allons montrer comment réaliser un programme simple pour créer une liste de notes d'étudiants.

Nous souhaitons que notre programme permette :

- d'entrer le nom d'un étudiant et sa note,
- de consulter la note d'un étudiant dont on connaît le nom,
- d'écrire une liste des noms et notes d'étudiant, triée par note.

Les opérations demandées peuvent être confiées à des fonctions dont l'interface est donné par le Programme 1.4.

L'interface utilisateur. Quand nous disposons des fonctions de traitement d'une feuille de notes il est possible d'écrire le programme d'interface utilisateur sans rien connaître des structures de données utilisées pour représenter une feuille de notes, ni le mécanisme des fonctions de traitement. Le programme 1.5 présente

Programme 1.5 Programme de notation (main)

```
// $Id: chap-intro.tex,v 1.2 2002/09/06 13:40:38 marc Exp $
#include "FeuilleNotes.hh"
#include <iostream>
int main(){
   using std::cout; using std::cin; using std::endl;
   cout << "Bonjour, vous êtes dans le programme "Notation ""<<endl;
   char reponse;
   bool sortie(false);
   while (!sortie) {
      cout << "voulez vous: "<<endl;
      cout << "    S : Sortir." <<endl;
      cout << "    E : Entrer une note." <<endl;
      cout << "    C : Consulter une note." <<endl;
      cout << "    L : Lister toutes les notes."<<endl;
      cin >> reponse;
      switch (reponse) {
         case 's':
         case 'S':
            sortie=true;
            cout << "Au revoir."<<endl;
            break;
         case 'e':
         case 'E':
            entrerNote();
            break;
         case 'c':
         case 'C':
            consulterNote();
            break;
         case 'l':
         case 'L':
            listerNotes();
            break;
         default:
            cout << "réponse " <<reponse <<" inconnue"<<endl;
            break;
      }
   }
   return 0;
}
```

Programme 1.6 données de la feuille de notes.

```
1 #include <string>
2 #include <iostream>
3 #include <vector>
4 namespace{ //namespace anonyme
5 struct Notation {
6       std::string etudiant;
7       float note;
8 };
9 typedef std::vector<Notation> FeuillesNotes;
10 FeuillesNotes feuilleNotes;
11 bool sale(false); //Booléen qui indique que le vecteur doit être trié.
12 } //fin du namespace anonyme
```

l'interface.

Les données Pour enregistrer une note d'un étudiant nous pouvons utiliser la structure `Notation` (Prog. 1.6 ligne 5).

Pour enregistrer une suite de notes nous aurons recours à un conteneur de la librairie standard. Pour commencer nous essayons d'employer un vecteur.

Les vecteurs nous fournissent une liste indexée de valeurs dont l'utilisation est similaire à celle des tableaux, mais ils évitent les problèmes de gestion mémoire des tableaux en `C`. C'est donc un type d'emploi aisé pour le programmeur qui connaît les tableaux, de plus il permet une insertion efficace en fin de tableau (opération `push_back`).

La déclaration du type des vecteurs de `Notation` se fait par la ligne 9 du programme 1.6 et un exemplaire initialement vide est déclaré ligne 10.

L'inefficacité de l'insertion en milieu de vecteur, nous conduit à entrer dans la feuille de notes les étudiants selon leur ordre d'arrivée, et à trier le vecteur ultérieurement. Nous utiliserons donc une variable `sale` (Prog. 1.6 ligne 11) qui indique que le tableau n'est pas trié.

Comme nous désirons que ces données ne puissent être vues que par les trois fonctions de l'unité de compilation courante, nous les plaçons dans un espace de nom anonyme (ligne 4), ce qui empêche que l'éditeur de lien exporte leurs noms. Nous savons dès lors que tout accès à ces données se fera par les trois fonctions précitées.

Entrée des notes Programme 1.7

L'entrée des notes diffère suivant qu'il s'agit d'une nouvelle note ou d'une modification de note. Pour savoir si un étudiant se trouve dans la feuille il nous

Programme 1.7 Feuille de notes (Entrée des notes)

```
13 void entrerNote(){
14    using std::cin; using std::cout; using std::endl;
15    std::string etudiant;
16    float note;
17    cout << "Entrez le nom de l'étudiant" << endl;
18    cin >> etudiant;
19    cout << "Entrez la note" << endl;
20    cin >> note;
21    FeuillesNotes::iterator  place;
22    for (place=feuilleNotes.begin(); // chercher la place de l'étudiant
23         place < feuilleNotes.end()&&place->etudiant!=etudiant;
24         ++place);
25    if (place < feuilleNotes.end()) { // trouvé dans la feuille
26       cout << "Note précédente de" << etudiant
27            << ":" << place->note <<endl;
28       place->note = note;
29       cout << "Nouvelle note " <<  ": " << place->note <<endl;
30     } else { // étudiant n'est pas dans la feuille
31        sale=true;
32        Notation notation={etudiant,note};
33        feuilleNotes.push_back(notation);
34        cout << "Nouvelle note de" << etudiant << ":" << note <<endl;
35    }
36 }
```

faut la parcourir.

Un conteneur se parcourt avec un itérateur (ligne 21) que l'on peut déplacer en l'incrémentant avec l'opérateur ++ (ligne 24) entre le début du conteneur (`begin()` ligne 22) et la fin (`end()` ligne 23) qui est située *après* le dernier élément du conteneur.

Si l'étudiant est dans la feuille, son ancienne note est `place->note`, elle est modifiée par affectation (ligne 28).

Sinon une nouvelle structure `Notation` est créée (ligne 32) et elle est mise en fin de vecteur par l'opération `push_back` (ligne 33). Notre feuille de note n'est plus maintenant triée, ce que nous indiquons par la ligne 31.

Remarquons que l' entrée des notes n'est pas vraiment robuste puisque la ligne 20 ne prévoit pas le cas ou l'entrée ne correspond pas à un `float`, ce qui ferait boucler le programme. Une solution est donnée §6.2.5 page 107.

Consultation des notes. Programme 1.8

La consultation se fait par parcours de la feuille avec la technique utilisée pour l'entrée des Notes.

Liste des notes. Programme 1.9

La liste des notes demande que la feuille soit triée. Si elle ne l'est pas déjà nous demandons à la fonction `sort` de la bibliothèque standard de trier le conteneur (ligne 53).

Pour que ce tri soit possible il faut spécifier comment sont ordonnées les notations, ce que fait l'opérateur < défini ligne 62

1.3 Les algorithmes

Dans la section précédente nous avons parcouru un vecteur pour y chercher et y insérer une valeur. Les techniques de recherche et d'insertion dans un conteneur sont toujours similaires, nous les répétons donc un grand nombre de fois ce qui représente un risque d'erreur. L'algorithme de parcours linéaire que nous avons choisi est compact, mais quand le conteneur est trié il est beaucoup plus lent qu'une recherche dichotomique.

Des algorithmes standards existent et sont disponibles pour tous les conteneurs de la STL, nos programmes gagneront en sécurité à les employer.

Nous avons déjà utilisé l'algorithme `sort` dans le programme 1.9 ligne 53 et la consultation des notes a été réécrite avec les algorithmes dans le programme 1.10.

L'algorithme `find` de la ligne 8 du programme 1.10 cherche la première occurrence d'un élément *égal* à la valeur indiquée qui se trouve entre deux itérateurs.

Alors que `find` peut être appliqué à tout conteneur `lower_bound` ne peut s'appliquer qu'à un conteneur trié. Il trouve aussi le premier élément *égal* à la valeur indiquée mais la recherche est dichotomique, donc beaucoup plus rapide.

Pour les deux algorithmes précédents l'égalité est testée avec l'opérateur `==` nous avons donc redéfini cet opérateur ligne 65 du programme 1.9 pour que la comparaison ne se fasse que par le nom d'étudiant.[1]

1.4 Le conteneur `Map`

Nous avons utilisé pour la feuille de note des vecteurs, mais nous devons reconnaître que ce conteneur n'est pas vraiment adapté à notre problème. Nous

[1]Ce programme a donc deux imperfections : l'opérateur `==` serait plus élégamment défini comme une méthode de classe (§7.6.8) , et il serait plus clair de laisser l'égalité comparer les deux champs d'une `Notation` et fournir pour l'algorithme un comparateur sous forme d'objet-fonction (**??** et 10.2.1)

Programme 1.8 Feuille de notes (consultation)

```
void consulterNote(){
   std::string etudiant;
   cout << "Entrez le nom de l'étudiant" << endl;
   cin >> etudiant;
   FeuillesNotes::const_iterator  place;
   for (place=feuilleNotes.begin();
        place < feuilleNotes.end()&&place->etudiant!=etudiant;
        ++place);
   if (place != feuilleNotes.end()) {
      cout << etudiant << ": " << place->note<<endl;
   } else {
      cout << etudiant << " est inconnu." <<endl;
   }
}
```

Programme 1.9 Feuille de notes (liste)

```
void listerNotes(){
   if (sale) {
      std::sort(feuilleNotes.begin(),feuilleNotes.end());
      sale=false;
   }
   for ( FeuillesNotes::const_iterator place(feuilleNotes.begin());
      place != feuilleNotes.end(); ++place){
      cout << place->etudiant << " : "<<place->note<<endl;
   }
}

bool operator < (const Notation & n1, const Notation& n2){
   return n1.etudiant < n2.etudiant;
}
bool operator == (const Notation & n1, const Notation& n2){
   return n1.etudiant == n2.etudiant;
}
```

Programme 1.10 Consultation avec les algorithmes standards

```
void consulterNote(){
   std::string etudiant;
   cout << "Entrez le nom de l'étudiant" << endl;
   cin >> etudiant;
      Notation recherche={etudiant,0.};
      FeuillesNotes::const_iterator place;
   if (sale) { //recherche dans tout le vecteur
      place = std::find(feuilleNotes.begin(),
                   feuilleNotes.end(), recherche);
   }else{ //recherche ordonnée (quicksort)
      place=std::lower_bound(feuilleNotes.begin(),
                         feuilleNotes.end(), recherche);
   }
   if (place<feuilleNotes.end() && place->etudiant==etudiant) {
      cout << etudiant << ": " << place->note <<endl;
   } else {
      cout << etudiant << " est inconnu." <<endl;
   }
}
```

Programme 1.11 Tableau associatif de notes (déclaration)

```
#include <map>
namespace{ //namespace anonyme
  typedef std::map< std::string, float> FeuillesNotes;
  FeuillesNotes feuilleNotes;
}
```

avons dû ajouter les nouveaux éléments en fin de vecteur, ce qui nous oblige à des tris périodiques, et alourdit le traitement puisque le conteneur est parfois trié et parfois ne l'est pas.

La STL nous propose le conteneur `map` qui permet de réaliser un tableau associatif. Avec la déclaration du programme 1.11 nous pouvons accéder à la note de `"toto"` par l'expression `feuilleNotes["toto"]`, l'entrée des notes est donc directe, elle est donnée programme 1.12 qui se comparera avec le programme 1.7. Quand à la consultation elle tire parti de la fonction membre `find` qui est une recherche rapide spécifique des `map`, elle est donnée programme 1.13.

Programme 1.12 Tableau associatif de notes (Entrée des notes)

```
void entrerNote(){
   std::string etudiant;
   float note;
   cout << "Entrez le nom de l'étudiant" << endl;
   cin >> etudiant;
   cout << "Entrez la note" << endl;
   cin >> note;
   feuilleNotes[etudiant]=note;
   cout << "Nouvelle note de" << etudiant << ":" << note <<endl;
}
```

Programme 1.13 Tableau associatif de notes (Consultation des notes)

```
void consulterNote(){
   std::string etudiant;
   cout << "Entrez le nom de l'étudiant" << endl;
   cin >> etudiant;
   FeuillesNotes::const_iterator place(feuilleNotes.find(etudiant));
   if (place != feuilleNotes.end()) {
      cout << etudiant << ": " << place->second <<endl;
   } else {
      cout << etudiant << " est inconnu." <<endl;
   }
}
```

1.5 Introduction aux classes et exceptions.

Nous avons utilisé dans le programme précédent la structure `Notation` qui comprend le nom de l'étudiant et sa note.

La note est déclarée comme un `float`, mais nous aimerions être certain qu'elle ne varie que dans les limites de l'échelle de notation (par exemple 0 à 10, ou 0 à 20, ou encore 0 à 100). Il se pose aussi le problème des étudiants qui n'ont pas pu être noté car ils n'ont pas rendu de copie. Cet état doit être enregistré à la place de la note, et nous pouvons employer pour cela, soit un booléen, soit une valeur spéciale de la note. Dans tous les cas nous voulons être sûr que la valeur de la note d'un étudiant non noté ne sera pas confondue avec une vraie note, par exemple qu'elle ne sera pas prise en compte pour le calcul des moyennes.

Comment assurer que ces demandes seront prises en compte dans tout le programme ? Une première réponse serait de dire qu'il suffit de s'assurer de la validité des notes dans la fonction d'entrée des notes. C'est une réponse raisonnable, le programme est petit, nous le connaissons dans son intégralité, et nous avons pris la précaution d'enfermer la structure `Notation` et la feuille de note dans un espace de nom anonyme qui assure que nous ne pouvons y accéder que par l'unité de compilation courante. Nous pouvons donc inspecter les deux cents lignes de code qui constituent cette unité de compilation et constater que l'on ne modifie les notes que dans la fonction d'entrée, on modifiera aisément cette dernière.

Considérons maintenant le problème dans un cadre un peu plus général. Supposons que l'on nous rapporte un programme de notation de quelques dizaines de milliers (ou même quelques milliers) de lignes de code, et que l'on nous signale que des notes non comprises dans l'intervalle 0-20 sont occasionnellement observées.

Comment trouver le code fautif ? L'inspection de tout le code est impossible de par sa taille. L'utilisation d'un débogueur ne nous serait possible que si nous savions reproduire l'erreur, et même alors elle conduit à un processus lent et sans certitude de succès.

Une seule chose peut nous aider c'est que le programme soit conçu de manière à établir une barrière claire autour des données, et qui ne permet leur accès que par des procédures sûres. Il est dans ce cas aisé d'isoler la faute, puisqu'elle ne peut résider que dans une des procédures à qui l'accès est autorisé. Bien entendu les modules extérieurs peuvent effectuer des requêtes incohérentes à ces procédures, mais c'est à elles que revient la responsabilité de les détecter et soit de les corriger, soit interrompre le traitement tout en signalant l'erreur à la procédure appelante.

1.5.1 Déclaration d'une classe.

Programme 1.14 Déclaration de la classe Notation

```
1 #include <iostream>
2 #include <string>
3
4 const float noteMin=0.;
5 const float noteMax=20.;
6
7 class Notation {
8    public:
9       struct HorsPortee{};
10      struct SansNote{};
11      Notation(std::string etudiantN);
12      void note(float noteN) throw ( HorsPortee );
13      float note()const throw ( SansNote );
14      bool estSansNote() const;
15      std::string etudiant()const;
16   private:
17      std::string etudiantM;
18      float noteM;
19 };
```

Les **Classes** donnent un moyen d'isoler un type de données et de fournir un jeu de fonctions qui permettent d'y accéder. Les **Exceptions** quant à elles permettent de signaler et de traiter les erreurs lors de l'accès aux classes. Les classes sont traitées en détail chapitre 7 et les exceptions §3.3.

En utilisant les classes nous réécrirons `Notation` comme indiqué dans le programme 1.14

1.5.2 Explication détaillée.

Parcourons ligne à ligne la déclaration de la figure 1.14.

l. 4, l. 5 Ces deux constantes signalent les limites permises pour une note. L'usage des constantes permet de ne pas avoir à changer toutes les sources quand ces limites changent. Cependant un changement des constantes oblige à recompiler tous les programmes qui utilisent cet en-tête (voir §2.5.2). Si les limites de la notation sont susceptibles de variations fréquentes ces constantes devront être remplacées par des méthodes de la classe `Notation`.

l. 7 Signale une déclaration de classe. `Notation` est le nom de la classe.

l. 8 `public` indique que ce qui suit est l'interface, c'est à dire la partie visible par l'utilisateur de la classe (voir §7.2.1).

l. 9, l. 10 Ces deux structures vides fournissent les noms des exceptions que peuvent lever les membres de la classe.

l. 11 Ce membre qui porte le même nom que la classe est un constructeur (voir §7.4.1). Les constructeurs servent à initialiser de nouveaux objets. Ainsi :

```
Notation n("toto");
```

est un nouvel objet de nom `"toto"` et sans note.

l. 12 Ce membre sert à affecter une nouvelle note à l'étudiant. Il est susceptible de lancer l'exception `HorsPortee`.

l. 13 Ce membre sert à accéder à la note de l'étudiant, il est susceptible de lancer l'exception `SansNote` si on essaie d'accéder à une note inexistante.

l. 14 Fonction booléenne qui indique si l'étudiant est sans note.

l. 15 Accesseur (§7.3) au nom de l'étudiant.

l. 16 Le mot clé `private` (voir §7.2.1) indique que les membres suivants sont privés, et qu'ils ne sont donc pas accessibles à l'utilisateur du module. Ils sont donc sans intéret pour lui, mais un fichier en-tête est à la fois destiné au programmeur et au compilateur. Ce dernier doit savoir quelle place réserver pour toute variable déclarée, information qu'il déduit de sa connaissance de l'intégralité des membres. C'est pourquoi cette composition interne des objets figure dans la déclaration de classe. L'utilisateur du module doit ignorer cette partie et ne pas utiliser les caractéristiques internes de l'objet.

l. 17, l. 18 Les deux membres qui permettent de représenter la notation.

1.5.3 Utilisation de la classe `Notation`

1.5.4 Explication détaillée.

Examinons maintenant le programme 1.15

l. 4 Cette ligne appelle le constructeur 1.14 l. 11 pour construire un nouvel objet de type `Notation` et l'initialiser (voir §7.4.1). Comme toute construction de variable *automatique* cet objet existe dans les limites du bloc englobant, c'est à dire ici le `main`.

l. 6 Il s'agit ici d'une boucle spéciale dite *infinie* ou plus exactement *boucle généralisée* la condition pour continuer la boucle est `true` et est donc toujours vraie, et ne sert pas à déterminer la fin de la boucle. La sortie de la boucle devra donc être déclenchée par un `break`. ce type de boucle est employé quand on ne peut déterminer la condition de sortie ni au début ni à la fin de la boucle. Elle peut être remplacée par une boucle qui teste une variable

Programme 1.15 Utilisation de la classe `Notation`

```
1 int main(){
2   using std::cout; using std::cin;
3   using std::endl; using std::cerr;
4   Notation notation("toto");
5   float nouvNote;
6   while(true) {
7     try{
8       cout << "Entrez la note de "
9       << notation.etudiant() << " : ";
10      cin >> nouvNote;
11      notation.note(nouvNote);
12    }
13    catch(Notation::HorsPortee){
14      cout<< "Note hors portée recommencez"<<endl;
15      continue;
16    }
17    catch( ... ){
18      cerr << "Exception inconnue"<<endl;
19      return 1;
20    }
21    break;
22  };
23  cout << "Au revoir"<<endl;
24  return 0;
25 }
```

booléenne servant à reporter en tête de boucle la condition(Les différentes boucles sont comparées §3.2.6).

l. 7 L'ordre `try` (§3.3) indique que l'on effectue le bloc entre accolades, en fournissant une série de gestionnaires d'erreurs qui seront appelés si une exception survient dans le bloc.

l. 11 Il s'agit là de l'appel de la fonction membre 1.14 ligne 12. Cet appel est constitué de trois parties :

`notation` situé avant le point, indique l'objet auquel s'applique la méthode (*méthode* est un autre nom pour *fonction membre*). Il s'agit de l'objet déclaré l. 4

`note` Donne le nom de la méthode. Ce doit être une méthode de la classe de l'objet situé avant le point. Comme `notation` est une méthode de la classe `Notation`, `note` désigne la méthode 1.14 l. 12.

`nouvNote` L'argument de la méthode, il s'agit de la variable déclarée l. 5.

l. 12 La fin du bloc `try` : à partir de ce point sont énumérés les gestionnaires d'erreurs disponibles.
- Si une exception est lancée et qu'un gestionnaire correspondant au type de cet exception est fourni il sera appelé.
- Si une exception est lancée sans qu'il y ait de gestionnaire correspondant le programme est arrêté avec un message d'erreur.
- Enfin dans le cas ou l'exécution du bloc n'a pas déclenché d'exception le traitement se poursuit après le bloc.

l. 13 Le premier gestionnaire d'erreur est destiné aux exceptions de classe `Notation::HorsPortee`, ce type d'exception sera effectivement lancé quand l'utilisateur entre une note qui n'est pas comprise entre `noteMin` et `noteMax`.

l. 14 Dans ce cas on affiche un message d'erreur à l'utilisateur et ...

l. 15 l'ordre `continue` demande de recommencer une itération de la boucle et donc de revenir l. 6.

l. 17 Ce second `catch` permet de gérer toutes les exceptions qui n'ont pas étés interceptées par un autre gestionnaire. Remarquons que si nous n'avons pas fourni de gestionnaire spécifique pour les exceptions `SansNote`, c'est simplement parce que le code de ce bloc `try` ne devrait pas lever de telles exceptions. De fait la seule exception *raisonnable* est `HorsPortee`, nous récupérons ici des exceptions inattendues. Bien sûr un programme plus achevé devrait aussi fournir un traitement pour les erreurs d'entrée-sortie.

l. 19 Comme nous ne savons pas comment poursuivre après une exception inconnue nous abandonnons le programme : pour cela le `return 1` renvoie la valeur `1` au shell et lui permet de savoir que le programme à détecté une erreur.

l. 21 La dernière ligne de la boucle est cet ordre `break` qui provoque ici une sortie *sans condition* de la boucle, autrement dit si une exception ne vient pas perturber son déroulement, cette boucle n'est exécutée qu'une seule fois.

l. 22 Cette accolade est celle de la fin de boucle.

l. 24 En cas de sortie sans erreur le `main` renvoie au shell la valeur `0`

1.5.5 Définition des méthodes de la classe notation.

l. 1 Cette valeur spéciale est employée pour marquer les notes non attribuées. Elle ne doit donc pas figurer parmi les notes possibles.

l. 3, **l.** 4 Cette fonction utilitaire permet de tester la validité de la note attribuée par la procédure `note`.

Programme 1.16 Corps de la classe `Notation`

```
1 const float pasDeNote=-1.;
2
3 inline bool noteOK(float note){
4    return note >= noteMin && note <= noteMax;
5 }
6
7 Notation::Notation( string etudiantN):
8    etudiantM(etudiantN),
9    noteM(pasDeNote){
10 }
11
12 void Notation::note(float noteN) throw ( HorsPortee ){
13    if ( ! noteOK(noteN)) throw HorsPortee();
14    noteM=noteN;
15 }
16
17 bool Notation::estSansNote() const {
18    return noteM==pasDeNote;}
19
20 float Notation::note()const throw ( SansNote ){
21    if  (estSansNote()) throw SansNote();
22    return noteM;
23 }
24
25 std::string Notation::etudiant()const{ return etudiantM;}
```

l. 7, l. 8, l. 9 Il s'agit du *constructeur* de la classe `Etudiant` le `string` passé en paramètre est conservé dans la *variable membre* `etudiantM` et la note de l'étudiant est initialisée avec la valeur spéciale déclarée l. 1.

l. 12 Il s'agit de la procédure qui permet de modifier la note d'un étudiant. Elle est déclarée comme susceptible de lancer une exception de type `HorsPortee`.

l. 13 L'exception est effectivement lancée quand la procédure est appelée avec une valeur qui ne constitue pas une note acceptable. Dans ce cas le flot de contrôle est interrompu et l'instruction suivante l. 14 ne sera pas exécutée. Au contraire le contrôle remontera successivement tous les blocs dans lequel est inclu l'instruction appelante jusqu'à ce qu'il trouve un gestionnaire pour le type de l'exception.

l. 14 Dans le cas *normal* l'affectation est effectuée ici.

l. 17, l. 18 Cette procédure nous permet de fournir un interface abstrait qui *cache*

la manière dont est codé le fait qu'un étudiant n'est pas encore noté. Elle permettra, par exemple, de changer à moindre frais le codage par valeur spéciale adopté ici, pour un codage par un booléen.

l. 20 La procédure qui permet l'accès à la note de l'étudiant, elle peut lancer une exception de type `SansNote`, et seulement une exception de ce type.

l. 21, **l.** 22 L'exception est lancée dans le cas où on essaie de connaître la note d'un étudiant *sans note*. La fonction ne peut donc pas être utilisée pour *savoir* si un étudiant est noté. L'utilisateur doit pour cela utiliser l'interface abstrait `estSansNote`.

Chapitre 2

Les types de données utilisés.

Les langages de programmation impératifs nous permettent d'effectuer des *actions* sur des *données*. Ces données peuvent être contenues dans des *variables* et transmises à des fonctions ou reçues d'elles. L'utilisation des variables est réservée aux langages *impératifs*, les langages *fonctionnels* se contentant comme leur nom l'indique, des fonctions. [1]

Nous allons utiliser un langage typé : les données manipulées sont pourvues d'un type, les fonctions sont accompagnées d'une signature qui précise les types de données qu'elles peuvent accepter ou rendre, et les variables sont étiquetées par le type des données contenues.

Nous allons tout d'abord examiner les mécanismes qui permettent de définir des objets typés, que cela soit des données, des variables ou des fonctions ; de préciser les noms et les lieux où ces objets sont connus, puis de définir de nouveaux types.

2.1 Déclaration et définition

Nous utilisons dans les langages de programmation des *noms* symboliques pour désigner les différentes entités que nous manipulons. Associer un nom à une entité et préciser son type de données et les lieux où elle est connue est la *déclaration*, réserver une place pour cette entité et lui donner une valeur initiale est la *définition*. Ces deux opérations peuvent être effectuées conjointement, ou l'une après l'autre.

[1] Les langages *logiques*, pour leur part ne connaissent rien de tel que fonctions ou données mais se contentent de transformer des assertions en d'autres assertions.

2.1.1 Le choix des noms

Rec. 1.1 Utilisez des noms significatifs.

La lisibilité du programme dépend pour une grande part du choix des noms, il est très important d'utiliser des noms significatifs qui indiquent de manière non ambigüe le rôle de l'entité désignée. Ce nom devra être compris non seulement par l'auteur au moment de l'écriture du programme, mais aussi par un programmeur qui entreprend la maintenance d'un programme inconnu réalisé depuis plusieurs années. Il convient donc de rejeter les sigles et abréviations excessives.

Cette convention implique aussi que l'on n'utilise jamais un objet pour deux usages différents, en particulier on évite les variables entières globales qui servent à des index différents tout au long d'un module. On pourra cependant utiliser des noms réduits à un ou deux caractères comme index de boucle si leur portée est très limitée.

Une convention de dénomination doit être utilisée pour l'ensemble des programmes développés par le service ou l'entreprise. Différentes conventions ou *styles* existent nous donnons celle que nous utiliserons dans les paragraphes marqués **Style**.

Rec. 1.2 Utilisez des noms anglais pour les identificateurs.

Dans le cas d'un programme qui peut être diffusé dans des pays étrangers ou dont les outils de développements, tels les compilateurs, sont d'origine étrangère on utilisera la langue anglaise pour les noms. C'est le seul moyen que les équipes étrangères qui maintiennent le programme le comprennent. N'oubliez pas non plus que si vous fournissez un rapport d'erreur pour un compilateur, le programme annexé devra être compris par un service de maintenance dont les membres sont souvent d'une nationalité différente de la vôtre.

Dans le cas ou certains membres de l'équipe de programmation comprendraient mal l'anglais cette règle se révèle inapplicable. En fait plus que la règle il faut prendre en compte la motivation, faciliter la compréhension du programme par tous ses utilisateurs, actuels ou potentiels.

Cependant dans le contexte de ce cours et des exercices nous pourrons utiliser des mots français.

Rec. 1.3 Soyez cohérents pour l'attribution de noms aux fonctions, types, variables et constantes.

Cette règle signifie que deux objets ayant des fonctions similaires doivent avoir des noms similaires. En particulier deux fonctions ayant le même rôle situées dans deux classes différentes devront avoir le même nom. Cette règle réduit de manière importante le temps nécessaire à la compréhension d'un programme complexe et les confusions à sa lecture.

Règle 1.8 Un nom ne doit pas comprendre deux barres de soulignement de suite.

Règle 1.9 Un nom ne doit pas commencer par une barre de soulignement.

Ces noms sont souvent réservés par les compilateurs, on évitera de les utiliser. Les conventions de style suivantes seront utilisées :

Style A.2 Quand un nom est composé de plusieurs mots tous les mots sont placés de manière contiguë et commencent par une majuscule, sauf éventuellement le premier.

Par exemple on écrira `squareRoot` comme nom de fonction.

Style A.3 Les noms des classes, `typedef`, et types énumérés commencent par une majuscule.

Style A.4 Les noms des variables et fonctions commencent par une minuscule.

Style A.8 Évitez les lettres qui se confondent avec des chiffres et réciproquement.

voir aussi les conventions de style A.5 p. 122 et A.6 p. 87 et le codage des classes paragraphe §7.2.2.

2.1.2 Déclaration

Tout nom utilisé dans un programme **C++** doit être déclaré. La **déclaration** définit un nouveau nom et précise au compilateur la **portée** c'est-à-dire la zone dans laquelle le nom est connu, et le type d'information que dénote ce nom.

En **C++** les déclarations peuvent être librement insérées parmi les autres instructions du programme alors qu'en **C**, elles doivent figurer en début de bloc.

2.1.3 Définition

Un nom doit faire l'objet d'une **définition** avant d'être utilisé. La définition d'un composant permet de réserver l'espace mémoire nécessaire pour stocker le composant, et de définir la valeur initiale de ce composant.

Alors que certains composants peuvent avoir plusieurs déclarations compatibles, il ne peut jamais y avoir plus d'une définition. Les exemples suivants sont des déclarations qui sont aussi des définitions :

```
extern const int marignan(1515);
int i;
int m(2);
char* const s("toto");
char t[5]="toto";
int v[]={1,5,1,5};
int triple (int  a){ return 3*a;}
struct St {int a; int b;};
enum Couleur { bleu, jaune, vert, rouge};
namespace N { int i;}
```

Les exemples suivants sont par contre des déclarations sans définition.

```
extern const int marignan;
extern int i;
int f(int&, int&);
struct St;
typedef int Taille;
using namespace N;
using N::i;
```

Ces noms se réfèrent à des variables définies par ailleurs, que cela soit dans la même unité de compilation ou dans un autre module objet ; à l'exception du `typedef` qui ne nécessite pas de définition car il ne représente qu'un synonyme pour un autre type.

Certaines déclarations comprennent une valeur initiale ou **initialisation** explicite : ce sont alors toujours des définitions. Pour les variables la présence d'un initialiseur ne suffit pas à différencier les déclarations des définitions, en effet :

- une déclaration de variable est toujours une définition quand elle n'est pas accompagnée du qualificatif `extern`,
- en revanche les déclarations `extern` ne sont des définitions que quand elles mentionnent une valeur initiale.

Une type (classe, énumération, union, structure), une fonction ou une constante gardent toujours la valeur initiale attribuée lors de leurs définitions. La valeur initiale d'une variable non constante peut être modifiée.

La déclaration d'un objet peut être répétée un nombre quelconque de fois.

Une unité de traduction (c'est à dire le code source compilé en une fois par le compilateur) ne peut pas contenir plus d'une définition par objet.

Un programme (constitué d'une ou plusieurs unités de traduction) doit contenir exactement une définition pour chaque objet, sauf dans le cas particulier des

fonctions en-lignes (qui doivent être définies dans chaque unité de traduction) et des fonctions patrons (qui peuvent avoir plusieurs définitions identiques)[2]

2.1.4 Initialisation.

La définition d'un objet peut mentionner un initialiseur qui précise au compilateur quelle doit être la valeur du nouvel objet.

Deux syntaxes sont possibles pour initialiser une variable `v` de type `T` avec une valeur initiale `init` :

```
T v=init;
T v(init);
```

Une variable d'un type prédéfini, dont la définition ne comprend pas d'initialisation spécifique est initialisée à une valeur par défaut quand elle est locale et statique, globale, ou définie dans un espace de nom. **Mais un objet local (appelé automatique) et un objet dynamique ne reçoivent aucune valeur initiale par défaut.**

L'exemple 2.1 montre quelques exemples de variables initialisées par défaut ou non initialisées. On notera qu'il y a une différence importante entre les types prédéfinis qui peuvent être non initialisés, et les objets de classe qui sont toujours construits avec un constructeur, que cela soit un constructeur par défaut ou un constructeur donné explicitement.

Cependant même les objets de classe peuvent contenir des zones non initialisées si ils comprennent, directement ou par transitivité, des membres de type prédéfinis non initialisés.

L'emploi d'une variable non initialisée étant une source très fréquente d'erreurs difficiles à corriger, on portera la plus grande attention à la recommandation suivante :

Rec. 5.2 Si possible initialisez les variables à l'endroit de leur définition.

Le compilateur utilise 0 comme valeur par défaut des types prédéfinis ; pour les types utilisateurs le compilateur utilise l'initialiseur par défaut fourni par le type, et en l'absence d'initialiseur par défaut, il reporte une erreur.

2.1.5 Portée

La partie du programme où un nom est connu est appelé la **portée** de ce nom. Un nom déclaré dans un bloc - c'est à dire dans une classe, une fonction ou un espace délimité par des accolades- est connu depuis sa déclaration jusqu'à la fin

[2] c'est aussi le cas des fonctions en-ligne avec une liaison externe

Programme 2.1 Exemple d'initialisation par défaut

```
#include <iostream>

int i1; // global
int Ti1[3];

int main(){
   using std::cout; using std::endl;
   int i2;
   int Ti2[3];
   static int i3;
   int* pi=new int;
   int* pT=new int[10];
   cout << "int i1 global: "<<i1<<endl;
   cout << "int Ti1[0] global: "<<Ti1[0]<<endl;
   cout << "int *pi sur le tas: "<<*pi<<endl;
   cout << "int pT[0] sur le tas: "<<pT[0]<<endl;
   cout << "int i2 local: "<<i2<<endl;
   cout << "int Ti2[0] local: "<<Ti2[0]<<endl;
   cout << "int i3 static local: "<<i3<<endl;
   return 0;
}
```

Résultat

```
int i1 global: 0
int Ti1[0] global: 0
int *pi sur le tas: 0
int pT[0] sur le tas: 0
int i2 local: 134521948       variable non initialisée
int Ti2[0] local: -1073745608 variable non initialisée
int i3 static local: 0
```

de son bloc de définition. Un tel nom est dit **local** et nous appelons **global** un nom situé hors de tout bloc. Un nom global est connu depuis sa déclaration jusqu'à la fin du fichier source où il figure.

Tout nom défini localement masque les synonymes globaux et ceux qui sont locaux à des blocs englobants. On peut toujours accéder aux noms globaux en les précédant du symbole :: comme illustré programme 2.2.

Programme 2.2 Accès à une variable globale.

```
int i(1);
int j(2);
int k(3);
int f ( int i, int j) { return (i+j+k);}
{
    int i(4);
    int k(i);
    int j (f( j, k));
    {
        int i(::i);
    }
    i =2;
}
int* p(&i);
```

Le masquage d'un nom par le nom le plus local peut permettre de développer des modules sans se soucier des noms déjà définis. Cependant l'oubli d'une déclaration peut provoquer un accès inopportun à un nom global précédemment défini.

Le mécanisme de masquage est parfois difficile à décrypter pour le lecteur humain et devient ainsi source d'erreurs persistantes.

On limite le risque d'accès à une variable synonyme ou une variable non initialisée en plaçant les variables dans le bloc local le plus interne. Ce qui implique que les variables sont à usage unique et qu'elles ne ont jamais réutilisées.

Les variables peuvent être directement initialisées avec des valeurs utiles ce qui évite les erreurs dues aux variables non initialisées ou initialisées avec une valeur aléatoire.

Rec. 5.1 Déclarez et initialisez les variables près de l'endroit ou elles sont utilisées.

2.1.6 Espace de noms.

Les règles précédentes nous permettent de limiter la portée d'un nom au bloc dans lequel il est utilisé. Cependant les objets externes tels que les constantes sta-

tiques, les classes, ou les fonctions globales gardent une portée elle aussi globale. Quand nous incluons un fichier d'en-tête nous ajoutons donc à notre espace tous les noms globaux de cette en-tête et tous les noms globaux des fichiers qu'elle inclut elle même.

Quand nous commençons notre programme nous ne disposons pas d'un espace vierge mais d'un espace déjà rempli par les différents paquetages que nous utilisons. Si certains de ces noms peuvent se référer à des entités que nous manipulons, la majorité d'entre eux ne nous sert pas et est inutile.

Ces noms inusités risquent de provoquer des erreurs de compilation quand ils entrent en concurrence avec des noms locaux et, plus encore, des erreurs d'exécution quand ils sont liés par inadvertance à des constantes ou fonctions que nous employons. Ces erreurs peuvent survenir lors de toute mise à jour d'un paquetage inclus.

Pour éviter ces problèmes les programmeurs de paquetages **C++** ont longtemps utilisés des classes artificielles pour grouper leur constantes globales, qui étaient déclarées comme variables statiques. Ce procédé ne permet pas de résoudre le cas des classes, fonctions et `typedef` globaux pour lequel on avait recours à la règle :

Rec. 15.12 Les noms globaux (classes externes, variables, constantes `typedef`s, et `enum`s) doivent être préfixés quand `namespace` n'existe pas sur le compilateur utilisé.

La solution sur les anciens compilateurs (avant les *espaces de nom*) est de préfixer tous les noms globaux d'un même paquetage par un court préfixe qui signe leur appartenance à ce paquetage.

La nouvelle norme admet la notion d'**espace de noms** qui s'écrit :

`namespace` *`identificateur`* `{` *`corps de l'espace de noms`* `}`

Tout nom défini dans un espace de noms peut être utilisé sans qualification dans cet espace, mais en dehors de cet espace il doit être préfixé par le nom de l'espace et l'opérateur `::`. Le programme 2.3 illustre l'accès aux noms d'un espace de noms.

Les espaces de noms rendent inutiles les identificateurs globaux.

Rec. 1.4 Les seuls noms globaux doivent être les identificateurs de `namespace`.

Un espace de noms peut être constitué de plusieurs déclarations `namespace` ayant le même identificateur.

Il est aussi possible d'avoir des espaces de noms sans identificateurs : le compilateur attribue à ces espaces un nom interne qui est différent pour chaque unité de traduction. On dit que ces espaces sont des *espaces de noms anonymes*.

Programme 2.3 accès aux objets dans un espace de noms.

```
#include <iostream>
int i(1);
int d(int i){ return 2*i;};
namespace Nme{
   int i(2);                         // nouvelle variable dans Nme
   int d (int i) {                   // nouvelle fonction
      return 3*i;                    // avec sa définition
   }
   int e (int i);                    // déclaration seulement
}

int main(int argc, char * argv[]){
  using namespace std;              // seulement dans main
   cout << i <<endl;                // => 1
   cout << d(i) <<endl;             // => 2
   cout << Nme::d(i) <<endl;        // => 3
   cout << Nme::e(Nme::i) <<endl;   // => 4
   return 0;
}

// définition de la fonction d de l'espace Nme
int Nme::e(int i){return i+2;};
```

On ne peut accéder aux noms définis dans un espace anonyme que dans la même unité de traduction.

L'accès et la définition de noms dans un espace de noms est illustré par le programme 2.4

Ainsi que le montre cet exemple une fonction peut être définie dans l'espace ou elle est déclarée ou bien en dehors de cet espace.

clause `using`

La clause `using` permet d'importer un nom d'un espace dans un autre espace de noms. C'est à dire qu'elle permet d'utiliser un nom externe à un espace de noms comme s'il figurait dans l'espace où est employé la clause `using`.

```
void f();
namespace A {
    void g();
    }
```

Programme 2.4 Définition dans un espace de nom.

```
namespace N {
    int i;
    int g(int a);      // déclaration de N::g(int)
    int k();
    void q();
}

namespace { int l(1); }// espace anonyme
namespace N {           // autre partie du namespace N
    char g(char a)       // surcharge N::g(int)
    int g(int a)        // définition de N::g(int)
    {
        return l+a;     // l vient de l'espace anonyme
    }
    int i;              // erreur i déjà défini dans cet espace
    int k();            // ok seconde déclaration de k()
    int k()             // ok: définition de N::k()
    {
        return g(i);    // appelle N::g(int)
    }
    int q();            // erreur essai de redéfinition de q
                        // avec un type de retour différent
}
char N::g(char a)       // définition de  N::g(char)
{
   return (a>='a' && a<='z')?a+'A'-'a':a;
}
```

```
namespace X {
    using ::f;    // f global
    using A::g;   // le g de A
    }
void h()
    {
    X::f();       // appelle ::f
    X::g();       // appelle A::g
    }
```

La clause `using` s'emploie de la même manière pour redéfinir les accès à une variable de l'intérieur d'une classe. (voir §8.2.3).

directive `using`

Contrairement à la clause `using` qui agit nom par nom, la *directive* `using` permet d'importer tous les noms d'un espace.

```
using namespace A;
g();              // A::g()
```

Rec. 1.5 N'utilisez pas de clause ou de directive `using` globales dans un fichier en-tête.

En effet une telle pratique réintroduit les noms globaux que les `namespace`s essayent d'éviter.

Par exemple on évitera :

```
//mon_module.hh
using namespace A;
using X::g;
inline void f(....){
  g();
}
.....
```

Bien que l'on évite ainsi de taper quelques caractères supplémentaire dans le fichier, cette pratique est très dangereuse. Si on peut parfois supposer que le lecteur de `mon_module.hh` peut se souvenir que tous les noms de `A` sont disponibles et que `g` se réfère à `X::g` ; il serait illusoire de penser que l'utilisateur qui met dans un fichier en-tête :

```
#include "mon_module.hh"
```

est conscient des changements qu'il induit dans son en-tête et qu'il propage chez tous les utilisateurs de son en-tête.

En revanche il est tout à fait possible de mettre ces `using`s dans un fichier implémentation C++, les clauses et directives `using` peuvent alors être utiles en évitant de répéter des chemins d'accès excessivement longs et ne nuisent pas à la clarté du code si le fichier est de taille modérée. Comme ce fichier n'est pas inclus dans d'autre fichiers on évite les problèmes de propagation des `using`s.

On pourra aussi utiliser `using` *même dans un fichier en-tête* à condition de s'assurer qu'il est placé *à l'intérieur d'un bloc*, cela pourra être utile pour les fonctions en lignes qui sont incluses dans les en-têtes par exemple :

```
//mon_module.icc
void f(....){
  using namespace A; //using limité à f()
  ....
}
```

Alias de nom d'espace

Il est possible de déclarer un synonyme à un espace de nom, dans une région d'un programme. cela se fait par l'ordre suivant :

```
namespace Un_espace_de nom_au_nom_tres_complet_mais_un_peu_long {
.....
}
namespace Bref = Un_espace_de nom_au_nom_tres_complet_mais_un_peu_long;
```

2.2 Durée de vie d'un objet

La durée de vie d'un objet ne doit pas être confondue avec sa *portée*, la portée marque seulement la visibilité de l'objet ; elle est une notion syntaxique, c'est à dire qu'elle constitue une zone du texte-source du programme. La durée de vie, quant à elle est une notion dynamique : elle représente l'espace de temps dans le déroulement du programme où la variable existe.

Tout objet a une durée de vie. Les objets globaux sont créés dans l'ordre de l' apparition de leur définition et pour la durée du programme.

Les objets locaux déclarés avec le mot clé `static` sont initialisés lorsque le flux de contrôle passe pour la première fois sur leur définition et ils existent jusqu'à la fin du programme.

Les autres objets locaux sont dits *automatiques* : ils sont construits chaque fois que le flux de contrôle passe sur leur définition, et détruits à la fin de leur bloc.

Les durées de vies de quelques objets sont illustrées programme 2.5

Nous reprendrons en détails les durées de vie des objets en §7.4.5.

2.3 Types de base

Tout objet défini dans le langage est associé à un type. Le type d'un objet détermine quelles opérations sont admises sur cet objet. Les types de **C++** peuvent être des **types de bases** ou des **types dérivés**.

Programme 2.5 Exemple de durées de vies.

```
#include <iostream>
using std::cout; using std::endl;
int i(13);
std::ostream s= cout << "bonjour"<< endl;
      //est éxecuté avant d'entrer dans le main()
main(){
    cout << "i="<<  i << endl;
    for (int j(1); j < 9;  *=  2){
         int i(j);
         static int k(j);
         i++;
         k++;
         cout << "i="<<  i << endl; // 2, 3, 5, 9
         cout << "k=" << k << endl; // 2, 3, 4, 5
    }
    cout << "i="<< i << endl;       //13 inchangé
//  cout << "k="<<  k << endl;  est illégal (hors portée de k)
    cout << "au revoir"<<  endl;
}
```

Les types de bases sont communs avec le langage **C** et permettent de représenter les pièces élémentaires d'informations que sont les caractères, les entiers, les nombres à virgule flottante.

Tous les types de bases permettent les opérations d'affectation et les opérations logiques d'*égalité* et *différence*.

Type vide

Le type `void` définit un ensemble de valeur vide. Il est utilisé comme type de retour des fonctions qui ne retournent pas de valeurs ; les pointeurs sur `void` sont aussi utilisés pour les pointeurs non typés.

2.3.1 Caractères

Les caractères sont représentés par trois types de même taille `char`, `signed char` et `unsigned char`. Ces trois types sont distincts. Le type `unsigned char` est un entier sans signe. Le type `signed char` est un entier signé, le type `char` peut être signé ou non suivant les systèmes. Ces trois types sont de la même taille (généralement un octet).

La taille du caractère est l'unité de mesure des places mémoires, aussi a-t-on `sizeof(char)==1`.

Les opérations définies par le système sur les caractères sont les opérations définies pour les types entiers.

2.3.2 Booléens

Le type `bool` est un type de base de **C**++ . Il a deux valeurs `true` et `false`. Il fait partie des types entiers. Il rend obsolète l'usage de **C** et dec premières versions de **C**++ d'utiliser un entier pour coder les valeurs booléennes.

2.3.3 Entiers

Les types entiers signés sont `signed short int`, `signed int`, et `signed long int`. Le mot `signed` ou le mot `int` peut être omis.

De la même manière toute déclaration où le type est omis est considérée de type `int`
Exemple :

```
const marignan(1515);
```

`marignan` est ici de type `int`.

Il est conseillé de ne pas utiliser ces raccourcis qui nuisent à la clarté du programme et se confondent avec les erreurs de frappe.

Les types non signés sont `unsigned short int`, `unsigned int`, et `unsigned long int` ; ils permettent d'accéder aisément à toutes les positions binaires d'un mot ou mot double. Les entiers non signés ne peuvent pas en revanche servir à s'assurer que le résultat d'une opération est positif. En effet les instructions :

```
unsigned int i;
i=-1;
```

provoquent seulement la conversion de `-1` en `fixed ent`.

Rec. 15.9 Quand c'est possible utilisez un simple `int` pour stocker, passer ou retourner une valeur entière.

Les `int` sont les plus appropriés et les plus efficaces, car ils correspondent généralement à la taille du mot mémoire, utiliser un `short` à la place d'un int ralenti les opérations, un `long` ne sera adopté que si il est vraiment nécessaire et les `unsigned` seulement pour des accès de bas niveau à la mémoire. On évitera ainsi les nombreux problèmes de conversion des entiers (cf. §2.9.1).

Le système définit sur les entiers les opérations suivantes :

Opérations arithmétiques : *addition*, *soustraction*, *multiplication*,*division*, *modulo*, *incrémentation* et *décrémentation*. (pré et post), *décalages* à droite et à gauche.

Opérations bit à bit : *et*,*ou inclusif*, *ou exclusif*, *complément*.

Opérations logiques : *et*, *ou*, *négation*, *opérateur conditionnel*.

Opérations de comparaisons (outre l'*égalité* et la *différence* déjà citées) : *plus petit*, *plus petit où égal*, *plus grand*, *plus grand ou égal*.

Affectations combinées aux opérations arithmétiques : Elles sont définies pour toutes les opérations arithmétiques et bit-à-bit.

Notons que les opérations arithmétiques sont aussi possibles sur les caractères, ce qui permet de les considérer comme des types entiers.

2.3.4 Nombres à virgule flottante

Les nombres à virgule flottante sont représentés par les types `float`, `double`, `long double`.

Les opérations sur les nombres à virgule flottante sont les opérations arithmétiques (à l'exception de l'opération *reste* `%` qui n'a pas de sens pour un réel) et les opérations de comparaisons.

2.3.5 Taille des types prédéfinis

Nous avons toujours

$$\mathtt{sizeof(shortint)} \leq \mathtt{sizeof(int)} \leq \mathtt{sizeof(longint)}$$

comme

$$\mathtt{sizeof(shortfloat)} \leq \mathtt{sizeof(double)} \leq \mathtt{sizeof(longdouble)}$$

mais ces inégalités ne sont pas toujours strictes.

Les valeurs des tailles des types prédéfinis sont définies dans l'en-tête `limits` par spécialisation du patron `numeric_limits`.

On pourra l'utiliser ainsi :

```
const int maxval = numeric_limits<short>::max()>100000 ?
                        100000:numeric_limits<short>::max();
```

Les principaux membres de la classe `numeric_limits` pour chaque type prédéfini *nombre* sont donnés figure 2.1.

Les constantes correspondantes en **C** sont décrite §5.4.1

Les limites numériques sont à considérer soigneusement dès qu'il y a des conversions implicites (voir §2.9.1).

numeric_limits<*nombre*>		
min()	*nombre*	valeur minimale
max()	*nombre*	valeur maximale
digits	int	nbre de bits
digits10	int	nbre de chiffres décimaux
is_specialized	bool	informations disponibles ?
is_signed	bool	nbre signé ?
is_integer	bool	nbre entier ?
is_exact	bool	représentation exacte ?
radix	bool	racine logarithmique
min_exponent	int	exposant binaire min
min_exponent10	int	exposant décimal min
max_exponent	int	exposant binaire max
max_exponent10	int	exposant décimal max

FIG. 2.1 – Membres de numeric_limits<*nombre*>

2.4 Littéraux

Un littéral est un objet constant dont le nom code la valeur. Le compilateur à la possibilité de décoder ce nom et de produire un objet de valeur correspondante.

Rec. 5.4 Les littéraux ne doivent être utilisés que pour la définition des constantes et des énumérations.

Il est recommandé de n'utiliser les littéraux que pour l'initialisation de constantes ou dans des énumérations. En effet si on peut déduire la valeur d'un littéral de son écriture, *le sens* de ce littéral lui n'apparaît pas, et il est courant de confondre deux littéraux de valeur identiques ainsi que de modifier par erreur un littéral dans une substitution globale d'une constante.

On écrira donc :

```
const int marignan(1515);
const int stockInitial(1515);
const int contenanceReservoir(1515);
....
volume = contenanceReservoir;
```

En revanche en **C**++ (*contrairement aux usages du langage C*), on évitera de placer les littéraux dans des macro-instructions pour représenter les constantes du programme. Les macros n'étant pas vues par le compilateur sont sources de nombreuses erreurs difficiles à détecter. On consultera à ce sujet le chapitre 4.3 et en particulier le paragraphe 4.3.1 et la Recommandation 13.5 p. 88.

2.4.1 Littéraux entier

Une suite de chiffres dont le premier est non nul constitue le codage décimal d'un entier.

```
1515
```

représente l'objet de type `int` dont la représentation décimale est `1515`. Un littéral entier est de type `int` si le type `int` est suffisamment grand pour le représenter, sinon c'est un `long` . Un message d'erreur est émis dans le cas ou le type `long` ne suffit pas à représenter le littéral.

Le littéral `0` est un `int` et les conversions permettent de l'utiliser pour tout entier, nombre en virgule flottante ou comme pointeur.

Une suite de chiffres hexadécimaux (0 ...9, *a* ...*z*) précédée de `0x` est le codage hexadécimal d'un entier. De même une suite de chiffres octaux précédée de `0` est le codage octal d'un entier. Ainsi :

```
17, 017, 0x17
```

représentent trois entiers de valeurs décimales `17,15,23`.

Des littéraux de type `unsigned int` peuvent être explicitement demandés en ajoutant le suffixe `U` à la constante. De même le suffixe `L` demande explicitement un `long int`. Ce qui est utile dans les appels de fonctions surchargées.

2.4.2 Littéraux en virgule flottante

Ils sont constitués d'une suite de chiffres décimaux représentant la partie entière du nombre, d'un point décimal, d'une suite de chiffre optionnelle représentant la partie fraction décimale, et éventuellement d'un exposant constitué d'un `e` ou `E` suivi d'un entier. Un littéral en virgule flottante provoque la génération par le compilateur d'un objet de type `double` qui est l' *approximation* de la représentation spécifiée. Nous pouvons aussi explicitement demander un type `float` avec le suffixe `f` ou `F` et un type `long double` avec le suffixe `l` ou `L`.
Exemple :

```
const double pi(3141.5659e-3);
```

2.4.3 Littéraux caractères

Un littéral caractères est constitué d'un caractère entre apostrophes. La constante `'a'` par exemple est l'objet de type `char` qui représente le caractère 'a' sur la machine employée. La valeur entière de `'a'` peut varier d'une machine à l'autre. Il

est aussi possible de spécifier directement en octal la valeur souhaitée d'un littéral `char` à l'aide d'un contre-oblique (\) suivi de un à trois chiffres octaux.

Le caractère dont la représentation interne est l'entier 48 s'écrira par exemple `'\065'`. Un littéral caractère peut être donné en hexadécimal en faisant suivre un contre-oblique de la lettre 'x' puis de chiffres hexadécimaux nous coderons donc aussi le caractère précédent, `'\x30'`.

Les constantes octales (resp. hexadécimales) sont terminées par le premier caractère qui n'est pas un chiffre octal (resp. hexadécimal). Bien que le langage ne l'impose pas, il est plus clair de toujours utiliser exactement trois chiffres octaux ou deux chiffres hexadécimaux pour les littéraux caractères.

Quand nous utilisons des codes mémoire des caractères nous obtenons des programmes non portables puisque le code des caractères varie suivant les systèmes.

Certains caractère spéciaux ont des noms symboliques prédéfinis qui utilisent le contre-oblique comme caractère d'échappement.

\n	passage à la ligne
\t	tabulation horizontale
\v	tabulation verticale
\b	retour arrière
\r	retour chariot
\f	saut de page
\a	alerte
\\	caractère contre-oblique
\? caractère ?	
\” caractère "	
\′ caractère ′	

2.4.4 Chaînes de caractères

Un littéral chaîne de caractères est constitué d'une suite de caractères entourée de guillemets. Un littéral chaîne de caractères définit un tableau de caractères qui contient les caractères spécifiés et le caractère additionnel `'\0'`. Les séquences d'échappement définies ci-dessus peuvent être utilisées dans une chaîne de caractères. La séquence `\n` permet d'inclure des changements de ligne dans une chaîne.

Exemple :

```
char s[]="ab\12345\n6"; //{'a','b','\123','4','5','\n','6','\0'}
```

2.5 Types dérivés

À partir des types connus il est possible de définir de nouveaux types à l'aide de constructeurs de type qui sont :

énumération

constante

référence

pointeur

tableau

fonction

classe (ou structure)

2.5.1 Énumération

Une énumération est un nouveau type entier. Les valeurs de ce nouveau type sont les constantes contenues dans l'énumération. Par exemple :

```
enum Jours { lundi, mardi, mercredi, jeudi,
              vendredi, samedi, dimanche};
```

définit un nouveau type `Jours`.

Les valeurs d'une énumération peuvent être converties implicitement en `int`. Par contre la conversion inverse doit être explicite.
Exemple :

```
Jours j(lundi);
int i(j);     // correct j==0
j=Jours(++i);// correct j==mardi
j=i++;       // erreur
```

Par défaut la première valeur est représentée par l'entier 0, la seconde par 1, et ainsi de suite ; il est possible à tout moment de donner explicitement un nouvel entier qui devient la valeur de la constante courante et le début de l'énumération qui suit.
Exemple :

```
enum Date { poitier=732, marignan=1515,
            anI=1792, anII, anIII, anIV};
Date d(anIV);
if ( d==anI+3){  //comparaison OK
  d=d+1;         //erreur écrire: Date(d+1)
}
```

2.5.2 Constantes

Le mot-clé `const` ajouté à un type le change en *type constant*. Un objet d'un type constant ne peut être modifié. Il est donc impossible de lui faire subir une affectation. On ne peut passer sa référence en argument à une fonction que si celle-ci déclare explicitement *explicitement* l'argument formel comme référence constante (cf. §2.6.2 et §3.4.2).
Exemple :

```
  const int marignan(1515);
```

définit une constante entière de nom `marignan` et de valeur 1515. Remarquons que, contrairement au langage **C** on n'utilise pas les macro-instructions pour définir des constantes en **C**++ . L'usage de types constants permet, contrairement aux macro-instructions, la vérification des types (cf. ci-dessus §2.4)

Puisqu'une constante ne peut être modifiée, sa valeur lui sera fournie à l'initialisation. Une constante peut éventuellement si elle est externe, ou membre d'une classe être déclarée sans que sa valeur ne soit donnée mais sa *définition* comprend toujours sa valeur définitive.[3]

La valeur d'une constante peut être une expression constante :

```
const int tailleEntier(sizeof(int));
```

ou une expression non constante :

```
  int i;
  cout << '?' << endl; cin >>  i;
{
    const int xxx(i);
  ..........
}
```

2.6 Référence

Nous appelons *identité d'un objet*, une propriété qui permet de l'identifier de manière unique. L'identité ne doit pas être confondue avec la valeur des membres de l'objet, où «état de l'objet» dont nous traiterons en §7.5.3, car dans les langages de programmation, *à la différence des bases de données*, deux objets distincts peuvent avoir la même valeur.

[3] Voir §2.1.3 pour la différence entre déclaration et définition, §7.4.1 pour l'initialisation des membres d'une classe, §7.3.3 pour les objets constants d'une classe.

Nous pouvons alors penser à l'identité comme à la place mémoire où se situe l'objet. Cette analogie très utile a ses limites, en effet le gestionnaire mémoire peut déplacer les objets durant leur vie, et bien qu'il ne puisse se trouver deux objets de même type au même endroit, un objet inclus dans un autre peut avoir la même adresse que lui.

L' identité d'un objet peut être obtenue par l'opérateur préfixé `&`. Quand l'objet possède un nom, le nom de l'objet désigne l'identité de l'objet, nous pourrions même simplement dire le nom *est* l'identité de l'objet. Un *type référence* permet de déclarer des alias pour les identités des objets du type de base. Une variable référence initialisée avec l'identité d'un objet définit donc un nouveau nom pour l'objet, qui sera synonyme des noms précédemment donnés à l'objet.
Exemple :

```
int i(1);  //i est le nom d'un entier de valeur initiale 1
int& j(i); //j est un nouveau nom pour i
++i;       //i (aussi appelé j!) vaut 2
++j;       //i vaut 3;
```

Toute référence doit être initialisée, il est impossible d'avoir un nom qui ne désigne aucun objet. La valeur initiale fournie doit être l'identité d'un objet existant (voir en §2.6.2 le cas particulier des références constantes), ainsi la déclaration suivante est illégale :

```
int& i=1; //erreur
```

Un objet doit exister *avant* d'être référencé et sa vie doit être au moins aussi longue que la référence, les règles de portées assurent un usage correct pour les objets automatiques. Mais pour le retour d'une référence par une fonction (cf. §3.4.3) ou pour une référence à un objet en mémoire libre le programmeur doit s'assurer que l'objet à une durée de vie qui excède celle de la référence.

Les références à des objets qui n'existent plus provoquent des erreurs souvent difficiles à détecter.

Exemple :

```
int* p(new int(1));  //pointeur sur l'entier 1
int& j(*p);          //référence à un entier de valeur 1;
j=2;                 //OK
delete p;
j=3;                 //erreur
```

On observe souvent de telles erreurs quand une classe change la place des membres qui assurent sa représentation mémoire, par exemple :

```
string s("toto et");
char& c(s[5] );
s+="lulu vont en bateau";
```

À la fin des instructions précédentes rien ne garanti que la référence `c` désigne toujours une lettre de `s`.

Les opérateurs n'agissent pas sur les références (i.e. pas sur les noms) mais toujours sur les objets désignés par les références.

Les références ne définissent pas un nouveau type d'objets, il ne peut donc pas exister de références à des références,de tableaux de références, ni de pointeurs sur des références.

L'utilisation des références comme arguments formels constitue leur principal usage. Elle est décrite en §3.4.2 et le retour d'une référence par une fonction en §3.4.3

2.6.1 Différence entre les références C++ et les références Java.

Les références utilisées en **C++** sont sensiblement différentes avec ce qui est aussi appelé références dans le langage Java. En Java une référence est toujours une indirection et est donc toujours représentée comme un pointeur, ce qui autorise des références nulles. En **C++** les références sont des synonymes et donc les références sont généralement traduites en l'adresse de l'objet correspondant par le compilateur, sauf dans le cas d'appel de fonctions qui ne sont pas en-lignes. Une référence correspond *toujours* à un objet existant et il n'y a pas de référence nulle.

En Java une variable sur un objet de classe contient une référence, c'est à dire un pointeur, sur l'objet situé en mémoire libre. En **C++** , à moins d'utilisation délibérée de *pointeur* et d'allocation mémoire, une variable contient l'adresse d'un objet sur le tas.

Si un objet est représenté par un bloc mémoire, en java sa copie et sa comparaison seront celles d'une référence à l'objet ; c'est à dire de l'adresse mémoire du bloc ; en **C++** il s'agit par défaut de la copie du bloc mémoire.

Cependant java et **C++** se rejoignent en ce qui concerne les types prédéfinis qui sont toujours traités par valeur.

2.6.2 Références constantes

Une déclaration de la forme :

```
const T& y(x);
```

déclare une référence constante `y`. On dit aussi «référence à une constante». Dans une référence constante, le mot clé `const` ne s'applique pas à l'identité de l'objet qui ne peut de toute façon jamais changer pour une référence, mais à l'objet référencé. Ici le mot `const` indique que je ne pourrai jamais changer la valeur désignée par `y`, sauf éventuellement en y accédant par un autre nom qui n'est pas déclaré `const`.

Certains objets peuvent avoir des références constantes et d'autres références non constantes ; en effet nous pouvons définir une référence constante avec un nom qui n'est pas déclaré constant. La référence constante donne alors un accès en lecture seule à cet objet.

En revanche les constantes ne peuvent être référencées que par des références constantes.

Les références constantes servent fréquemment comme arguments des fonctions, voir §3.4.2. Pour faciliter cet usage il est permis d'initialiser une référence constante (et seulement quand elle est constante) avec le résultat d'évaluation d'une expression quelconques.

Par exemple :

```
const int & i(1)
```

Dans ce cas une valeur temporaire de même type que la référence est créée, elle est initialisée avec l'expression (après conversion si son type est différent) et l'identité de cette variable temporaire sert à initialiser la référence. La déclaration précédente est donc équivalente à :

```
const int i(i);
```

Les initialisations de ce type sont utilisées pour le passage des arguments.

2.7 Pointeurs

Le type `T*` contient des pointeurs sur le type `T`, c'est à dire des **variables** qui peuvent contenir l'identité d'un objet de type `T`. Alors que toute occurrence d'un nom d'objet ou d'une référence est considérée par le compilateur comme représentant l'objet nommé, les pointeurs doivent subir une opération spéciale appelée *déréférence* ou *indirection* pour extraire l'objet pointé. Cette opération est dénotée par l'opérateur préfixé $*$; donc pour un pointeur `p`, `*p` est l'objet pointé par `p`.

En **C**++ nous nous servirons des pointeurs comme itérateurs [§9.1] sur des tableaux, c'est à dire sur une suite contiguë d'objets de même types. Le pointeur sera l'exemple générique des itérateurs d'accés direct (cf. §40)
Exemple :

```
int t[]={1, 2, 3, 4, 5, 6 ,7, 8 ,9 ,10};
int * p1=t+5;   // pointeur sur l'entier contenant 6
int * p2=p1-1;  // pointeur sur l'entier contenant 5
int * p3=p2+10; // Danger ne pointe pas sur un objet défini
```

Cependant le langage **C** fait un grand usage des pointeurs pour compenser l'absence de références, usage que nous pouvons trouver dans la programmes **C** ou traduits de ce langage.
Exemple :

```
int i, j;
int * p(&i);              //pointeur sur i
int * q(p);               //autre pointeur sur i
int * * r(&p);            //pointeur sur p
*r= &j;                   // p pointe sur j
p= q;                     // p pointe de nouveau sur i
char * s;                 // pointeur de caractères
int (*t)[7];              // pointeur sur un tableau de 7 entiers
int & (*pf)(const char *);// pointeur sur une fonction
                          // avec un argument const char*
                          // et une valeur de retour int&
int* k1, k2, k3[5];       // k1 est un pointeur sur un entier,
                          // k2 est un entier (pas un pointeur!),
                          // k3 est un tableau de 5 entiers.
```

La déclaration de `k1`, `k2`, `k3`, dans une même instruction est source de confusion, on évitera ces définitions groupées.

Rec. 5.3 Ne déclarez qu'une variable par instruction de déclaration.

2.7.1 Pointeur sur une constante.

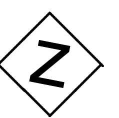
Nous pouvons avoir des pointeurs sur des types constants : `const T* p` déclare `p` comme pointeur sur une constante de type `T`. Cela implique que `*p` ne peut pas être changé, mais cela n'interdit pas de modifier la variable `p`.

Des pointeurs constants peuvent aussi être déclarés par `T* const p`; un tel pointeur ne peut être modifié mais, si le type `T` est non constant, `*p` peut être changé. Bien sûr il est aussi possible de déclarer des pointeurs constants sur des constantes.
Exemple :

```
const int i1(10), i2(3);
int j1(i1), j2(i2);
```

```
const int * p(&i1);      // pointeur sur une constante entière
int * const q(&j1);      // pointeur constant sur un entier
const int * const r(&i1);// pointeur constant sur une constante
p=&j2;      // p pointe sur j2;
*p=5;       // erreur on ne peut pas changer la valeur pointée
            // par ce pointeur déclaré sur une constante
p=q;        // erreur q n'est pas un pointeur sur const int
*q=5;       // j2==5
q=&j2;      // erreur q est constant
*r=5;       // erreur r pointeur sur const;
r=&i2;      // erreur r constant;
j2=*r;      // j2==10
```

2.8 Tableaux



Une définition d'un tableau de type `T` de la forme `T tab[taille]` réserve en mémoire la place pour n éléments de type `T` indicés de `0` à `taille-1`.Le nom `tab` est converti dans toutes les opérations [4] en pointeur sur le premier élément du tableau. La taille `taille` du tableau doit être une *expression constante*, c'est à dire une expression dont les membres sont des littéraux, ou des constantes entières initialisées avec des littéraux. Exemple :

```
char s[3];     //trois caractères s[0], s[1] et s[2]
int * a[2];    //tableau de deux pointeurs sur des entiers
int (*a)[2];   //a est un pointeur sur un tableau de 2 entiers
float v[2][5];//v[0] et v[1] sont des tableaux de 5 floats
```

Si `tab` est un tableau, pour toute valeur entière `n` l'expression `tab+n` est un pointeur sur son n+1-ième élément, si celui-ci est défini. Le n+1-ième élément peut être désigné par l'expression `tab[n]` qui est synonyme de `*(tab+n)`.

Plus généralement si `p` est un pointeur sur un élément d'un tableau et `n` un entier, alors `p+n` pointe sur le n-ième élément suivant p. Le résultat est indéterminé si le tableau ne contient pas suffisamment d'éléments. De même si `p` et `q` sont deux pointeurs sur un même tableau `q-p` est le nombre d'éléments à ajouter à `p` pour obtenir `q`, on a donc : `p+(q-p)==q`.

[4] sauf `sizeof` qui est directement interprété par le compilateur

2.8.1 Initialisation des tableaux

Un tableau peut être initialisé à l'aide d'une liste d'initialiseurs constituée d'une liste d'expressions constantes, du type des éléments, placées entre accolades. Quand un tableau est ainsi initialisé on peut omettre sa dimension, le compilateur la déduira alors de la dimension de l'initialiseur.

Dans ce cas particulier d'initialisation de tableau seule la syntaxe avec le signe = est autorisée, nous ne pouvons pas utiliser une valeur initiale entre parenthèses.

Dans le cas particulier d'un tableau de caractères nous pouvons aussi initialiser le tableau avec un litéral chaîne de caractères.

Si la liste d'initialiseurs ne fournit pas suffisamment de valeurs, le tableau est complété avec l'initialiseur par défaut (quand celui ci existe) ; quant aux valeurs en excès, elles sont ignorées. Exemple :

```
char s[]={'a','z','e','r','t'};
chat t[]="azert";        //comme ci-dessus avec '\0' à la fin
int v[4][5]={{1,2,3,4,5},{6,7,8,9,10},
       {11,12,13,14,15}};// v[3] est initialisé a {0,0,0,0,0}
```

2.8.2 Utilisation des tableaux

L'utilisation des tableaux en argument de fonction est délicate, très délicate quand les tableaux sont multidimensionnels, elle est traitée en §5.1. Les tableaux n'ont pas un protocole défini par le langage qui permette un accès sûr à leurs éléments ni leur transmission comme argument. En **C**++ un tableau devrait être considéré comme une représentation de bas niveau, et son utilisation être encapsulée dans une classe. Pour communiquer if faut utiliser une *classe tableau* qui gère la cohérence de la représentation des données (Voir Rec. 13.6 page 75). C'est en particulier vrai pour les tableaux de caractères ; un pointeur de caractère est généralement un bien mauvais moyen de communiquer une chaîne de caractères (à moins qu'elle soit statique et constante). Il est utile de se servir d'une *classe chaîne de caractères* qui peut gérer l'allocation et la récupération mémoire.

La bibliothèque standard fournit des classes patrons de tableaux `vector` §10.1 et des classes chaîne de caractères `string` §11

2.9 Conversions de type

2.9.1 Conversions implicites

Une fonction peut dans certains cas accepter un argument effectif d'un type différent de l'argument formel. L'argument effectif est alors converti dans le type

de l'argument formel lors de l'appel de la fonction.

Dans les cas où il s'agit de deux types numériques, **C++** procède à une conversion implicite de l'argument . Il en est de même pour les pointeurs dans les conditions citées plus loin, pour un objet d'une sous classe `public` de la classe attendue , ou si un opérateur de conversion est défini. .

Rec. 6.1 Préférez les conversions explicites aux conversions implicites.

Les conversions implicites rendent le programme peu lisible et en compliquent le débogage car elles ne sont pas portées dans le texte source et passent facilement inaperçues. Elle viennent aussi contrer le mécanisme de typage et rendent l'identification des fonctions appelées difficiles.

Les conversions implicites des types prédéfinis sont :

- Les `bool` peuvent être convertis en entier, `true` devient 1 et `false` devient 0.
- Les expressions arithmétiques, énumérations et pointeurs peuvent être converties en `bool` 0 devient `false` et toute autre valeur devient `true`.
- Les `bool`, `char`, `short int`, et les objets de types `enum` peuvent être utilisés à la place des entiers et sont convertis en `int`.
- Les entiers positifs sont convertis en entiers non signés de même valeur, un entier négatif $-n$ est converti en $2^p - n$, où p est le nombre de positions binaires.
- Les entiers non signés se convertissent dans l'entier correspondant quand ils sont inférieurs à la taille maximale de l'entier, sinon le résultat est indéterminé.
- Les différents types en virgule flottante sont convertis entre eux tant que la valeur maximale du type but n'est pas dépassée ; il y a éventuellement une perte de précision.
- Les flottants qui peuvent être représentés dans le type entier cible sont convertis en entiers. Ils sont alors tronqués [*INCITS* 4.5] , [5] le résultat de la conversion d'un flottant trop grand pour être représenté n'est pas défini.
- les entiers se convertissent en flottants avec éventuellement une perte de précision.
- Dans des opérations arithmétiques qui font intervenir deux opérandes de précisions différentes le moins précis est converti dans le type du plus précis.

[5] Si on désire un arrondi il faut le calculer éventuellement avec une fonction de bibliothèque. La `libc` `<cstdlib>` offre dans l'espace `std` les fonctions de la famille `ceil`, `floor`, `trunc`, `rint`, `round`, ces fonctions **C** permettent différents types d'arrondis. Elles ont des noms différents suivant le type d'argument.

Nous envisagerons les conversions d'objets de classe et le polymorphisme des opérations qu'elles permettent de réaliser dans le chapitre §8 sur l'héritage.

Règle 15.10 Assurez-vous que vous ne tronquez pas des données significatives dans une conversion vers un type plus petit.

De manière générale il faut être extrêmement prudent quand aux conversions qui risquent d'occasionner des troncatures, d'autant plus que la perte d'information occasionnée dépend de la représentation physique des types sur la machine employée. Ces conversions ne sont pas portables.

Ainsi une simple déclaration comme :

```
short i(100000);
```

est potentiellement dangereuse car rien ne garantit que les `short int` peuvent représenter l'entier `100000` sur toutes les machines. Nous suivrons donc la recommandation 15.9 (§2.3.3) en employant `int` autant que possible, dans le cas ou un autre type devrait être employé on suivra :

Règle 15.7 Ne supposez pas que les objets ont une taille ou une disposition arbitraire en mémoire.

Rec. 15.11 Utilisez des `typedef`s ou des classes pour cacher les types de données qui dépendent de l'application ou de la représentation machine.

On isolera donc les considérations dépendant de la machine dans un module

```
// Déclarations dépendant de la machine
#include <limits>
static const int  maxPetit(1000000);
typedef short petitEntier; // Nombres au moins jusqu'à  maxPetit
assert(maxPetit < numeric_limits<petit_entier>::max());
.........

petitEntier i(100000);
```

Conversions de pointeurs.

Le littéral `0` peut être converti en pointeur d'un type arbitraire, ce pointeur est toujours différent des pointeurs sur des objets. Un usage courant en **C**++ est de définir `NULL` comme une macro-instruction de valeur 0 et de l'utiliser comme pointeur nul universel.

Un pointeur sur un objet non constant peut être converti en `void*` et un pointeur sur un objet constant en `const void*`. `void*` n'est pas converti implicitement en un autre type de pointeur. C'est pouquoi la convention **C** de définir `NULL` comme `(void*)0` n'a plus de sens en **C**++ , ou en revanche, le littéral `0` joue le rôle de *zéro universel*.

Règle 6.3 Ne supprimez pas un `const` par une conversion.

Un pointeur de type `T*` peut être converti implicitement en `const T*` en restreignant ainsi les droits d'accès à l'objet. L'opération inverse ne peut être réalisée que par une conversion explicite. Comme il est anormal de donner plus de droits d'accès que ce qui nous est fourni l'*effacement* d'un `const` est généralement à prohiber.

Les données de classes qui sont modifiables bien qu'appartenant à un objet constant seront marquées `mutable` (voir §7.3.3) et ne seront pas l'objet de transtypage. Les quelque rares transtypage avec effacement de `const`, qui sont situés dans des modules de bas niveau, se feront avec l'opérateur `const_cast`.

2.9.2 Conversions explicites

Nous pouvons demander explicitement la conversion vers un type `T` par la notation `(T)obj` ou `T(obj)`. La seconde notation ne s'applique que si `T` est un nom simple de type. Nous pouvons donc écrire `(int)x` ou `int(x)` mais seulement `(int*)px`. Il est toujours possible de convertir ainsi un type de pointeur dans un autre type de pointeur, mais ce court-circuit du contrôle des types, conduit souvent à des constructions qui dépendent des mécanismes d'adressage de la machine utilisée et qui ne sont donc pas portables (cf. §2.9.1). Cependant on peut de manière sûre convertir un pointeur du type `T*`, d'abord vers le type `void*`, puis à nouveau en `T*`.

2.9.3 Les opérateurs de conversion

Pour limiter les dangers liés aux conversions tant explicites qu'implicites de nouveaux opérateurs de conversion ont été définis (*Groupe de travail ISO WG21 1995*). Ils permettent de vérifier dynamiquement ou statiquement la validité de la conversion des pointeurs, d'annoncer que l'on fait une conversion dépendante de l'implémentation, ou encore de demander de changer le qualificateur d'un type.

Leur utilisation, plus sûre, doit être préférée aux conversions implicites et aux anciennes conversions explicites.

Rec. 6.2 Utilisez les nouveaux opérateurs de conversion (`dynamic_cast`, `const-_cast`, `reinterpret_cast`, et `static_cast`) à la place des conversions dans l'ancien style, à moins que cela ne pose des problèmes de portabilité.

Opérateur `static_cast`

L'opérateur `static_cast` est utilisé quand la vérification est possible à la compilation. Il permet de rendre explicite des conversions autrement implicites et d'effectuer les conversions réciproques.

Les seules conversions autorisés par un `static_cast` sont :

1. Les conversions qui peuvent être effectuées de manière implicite.
2. Les conversions vers `void`. (Elles oublient simplement la valeur fournie).
3. Quand les pointeurs sur `T1` peuvent être convertis par `static_cast` en pointeurs sur `T2` alors l'identité d'un objet de type `T1` peut aussi être converti en référence à `T2`.
4. Les entiers peuvent être convertis en énumération.
5. `static_cast` peut réaliser l'inverse des conversions standard.
6. Les conversions *réciproques* de classe fille vers les classes mères ou de membre de classes filles vers les membres de classe mères à condition qu'elles soient vérifiables statiquement et ne provoquent pas la perte d'un qualificateur de type.

En revanche il est impossible d'éliminer un `const` avec un `static_cast`.

Opérateur `dynamic_cast`

L'opérateur `dynamic_cast` est utilisé pour convertir une valeur *dont le type est ignoré lors de la compilation*, i.e. pour les conversions de pointeurs ou références vers des classes de base.

L'opérateur `dynamic_cast` vérifie *lors de l'exécution* que le transtypage est possible sinon il rend la valeur nulle pour un pointeur ou lance une exception pour une référence.

Nous effectuerons cette opération :

1. Pour convertir un pointeur `v` dans un autre type pointeur `T`.
2. Pour récupérer l'identité d'un objet `v` dans un type référence `T`.

Le résultat de l'expression `dynamic_cast<T>(v)` est la conversion de la valeur `v` dans le type `T`.

Il est impossible d'éliminer un `const` avec un `dynamic_cast`.

2.10 Mémoire libre



2.10.1 Allocation *à la demande* de ressources mémoire.

Les objets *automatiques* sont alloués automatiquement lors de leur définition et désalloués à la sortie de leurs blocs de définition. Les objets *statiques* sont alloués pour toute la durée du programme. La *déclaration* de ces objets fournit un nom qui permet de les désigner.

Pour les objets en mémoire libre, le programmeur contrôle les créations et les destructions, il est responsable de conserver un chemin d'accès à ces données jusqu'à leur destruction[6].

L'opérateur `new` sert à créer à la demande un objet. L'instruction `new T(init);` crée un objet de type `T`, l'initialise avec la valeur initiale `init`, et renvoie l'identité de l'objet créé.

De tels objets existent jusqu'à ce que leur destruction soit demandée explicitement par le programmeur à l'aide de l'opérateur `delete`. Celui-ci doit toujours s'appliquer à l'identité d'un objet créé par `new`.

Règle 8.1 `delete` doit être utilisé seulement avec `new`.

Les tableaux d'objets en mémoire libre seront créés par `new[]` et détruits par `delete[]`.

Règle 8.2 `delete[]` doit être utilisé seulement avec `new[]`.

Dans le cas où nous avons créé un tableau, nous devons le détruire par l'appel de l'opérateur `delete[]`. Si nous contrevenons à ces règles nous risquons de ne pas initialiser ou détruire convenablement les objets.

Les primitives de **C**++ construisent et détruisent les objets qu'elles allouent en mémoire centrale. Les primitives de **C** effectuent la gestion mémoire mais ne se n'interviennent pas dans la construction et la destruction.

Règle 13.1 Utilisez `new` et `delete` à la place de `malloc` `calloc` `realloc` et `free`.

`free` ne peut pas (*toujours*) récupérer la place allouée par `new`, il n'appelle pas les destructeurs des objets dont la place est récupérée. `delete` ne sait pas récupérer la place allouée par `malloc`. Quand à `realloc` il sera remplacé par l'utilisation des conteneurs standards qui savent augmenter dynamiquement leurs tailles. On évitera totalement ces constructions en **C**++ .

---

[6]D'autres langages à la suite de LISP examinent périodiquement la mémoire pour supprimer les objets auquels on ne peut plus accéder : un tel processus appelé *ramasse-miettes* est absent de **C**++ (mais peut être fourni par certains paquetages)

Règle 8.3 N'accédez pas par un pointeur ou une référence à l'adresse d'un objet libéré.

Les pointeurs qui restent affectés à des adresses d'objets libérés sont des sources fréquentes d'erreurs difficiles à déceler, car changeant d'une exécution à l'autre. On les évitera en déclarant les pointeurs dans le bloc même où l'objet existe, quand l'existence d'un objet se confond avec celle d'un bloc syntaxique ; sinon nous réaffecterons immédiatement le pointeur devenu libre à `0` ou à l'adresse d'un autre pointeur.

Exemple :

```
....
{
  int * const i(new int(7));
  {
   int * const pt(new int[5]);
   ....
   delete[] pt;
   }
  ...
  delete i;
}
```

Il sera toujours plus sûr de se fier à une classe *pointeur intelligent* qui désactive les pointeurs sur les objets détruits. La bibliothèque standard offre les `auto_ptr` appropriés aux objets avec un unique propriétaire. De même la bibliothèque `BOOST` offre les `scoped_ptr` et `scoped_array` pour un propriétaire unique et `shared_ptr` et `shared_array` pour le partage de propriété. On pourra aussi consulter notre classe pointeur du programme 7.12 et la section 7.6.5

En ce qui concerne les pointeurs sur des collections d'objets (tableaux), ils seront remplacés par l'utilisation des conteneurs de la STL en particulier les vecteurs [§10.1].

Un objet créé en mémoire libre et qui n'est pas membre d'une classe n'est initialisé que dans le cas ou un *initialiseur* est présent. Si il n'y a pas d'initialiseur, et dans le cas des tableaux, la valeur initiale est indéfinie. En revanche les objets de classe sont toujours initialisés par un constructeur : soit une valeur initiale est demandé explicitement, ou sinon le constructeur par défaut est utilisé. Pour les tableaux d'objets de classe le constructeur par défaut (voir §7.4.3) est appelé pour chaque élément.

2.10.2 Épuisement des ressources mémoire.

Quand l'opérateur `new` ne peut pas trouver la place mémoire nécessaire pour allouer l'objet en mémoire libre il appelle un gestionnaire d'erreur qui par défaut lance l'exception `std:bad_alloc` définie dans `<new>`.

Une version qui se contente de rendre un pointeur égal à `0` sans abandon du programme est aussi disponible sous la forme :

```
new(nothrow) objet
```

Ce comportement implique que le programmeur doit se charger du contrôle de l'épuisement des ressources mémoires. Il peut le faire en fournissant un traitement approprié à l'appel du `new`.

On peut aussi remplacer le gestionnaire standard d'erreur par une autre fonction. Celle-ci sera une fonction sans argument ni valeur de retour c'est à dire du type :

```
 typedef void Fvoid_t();
```

On indique la fonction à employer en passant son identité à la fonction prédéfinie `set_new_handler` qui rend en retour l'identité du gestionnaire qui était défini auparavant.

```
Fvoid_t * set_new_handler(fvoid_t *);
```

Chapitre 3

Instructions

3.1 Opérateurs

Les opérateurs s'associent en suivant leur ordre de précédence ; les opérateurs ayant la même précédence s'associent généralement à gauche c'est à dire que `a+b+c` signifie `(a+b)+c`. Les opérateurs unaires et d'affectation, en revanche, s'associent à droite donc `a=b=c` signifie `a=(b=c)`. Une liste d'opérateurs classés par ordre de précédence est donnée par les tables 3.1 et 3.2.

Nous ne pouvons pas *a priori* supposer la commutativité ou l'associativité des opérateurs. Même pour les types entiers, les éventuels dépassements de capacité font qu'un résultat peut dépendre de l'ordre d'évaluation des opérateurs. Quant aux flottants, les approximations rendent leurs opérateurs arithmétiques non associatifs. Les opérateurs bits-à-bits sont commutatifs et associatifs tant que les sous-expressions impliquées n'ont pas d'effet de bord, alors que les opérateurs logiques sont explicitement non commutatifs.

Pour les opérateurs surchargés, l'utilisateur est libre de leur donner la signification qu'il souhaite, il est de sa responsabilité de les concevoir d'une façon qui ne contredise pas les habitudes, et de fournir une spécification de leur sémantique.

Il est utile de parenthéser les expressions, soit pour forcer un ordre d'évaluation, soit pour expliciter l'ordre d'évaluation quand celui-ci est obscur. On prendra garde en particulier à des expressions telles que `1<x<5` ; elle est équivalente à `(1<x)<5` et est donc toujours vraie.

Dans une expression (excepté pour l'opérateur *virgule* et les opérateurs logiques, cf. paragraphe suivant) l'ordre d'évaluation des sous-expressions n'est pas défini. Il faut donc éviter des expressions comme `j=2*i+i++` dont le résultat dépend de l'ordre d'évaluation.

résolution de portée	*classe*::*membre*
résolution de portée	*espace_de_nom*::*membre*
nom global	::*nom*
sélection de membre	*objet* . *membre*
déréférence et membre	*pointeur*-> *membre*
indexation	*pointeur*[expr]
appel de fonction	*fonction* (*liste-expr*)
construction de valeur	type (*liste-expr*)
post-incrémentation	*objet* +mutt+
post-décrémentation	*objet* --
identification d'un type	typeid *type*
type d'une expression	typeid *expr*
conversion à l'exécution	dynamic_cast< *type* >(*expr*)
conversion à la compilation	static_cast< *type* >(*expr*)
conversion non vérifiée	unchecked_cast< *type* >(*expr*)
oubli d'un const	const_cast< *type* >(*expr*)
taille d'un objet	sizeof *expr. const.*
taille d'un type	sizeof(*type*)
pré-incrémentation	++ *objet*
pré-décrémentation	-- *objet*
complément	~ *expr*
négation	! *expr*
moins unaire	- *expr*
plus unaire	+ *expr*
identité	& *objet*
déréférence	* *pointeur*
allocation mémoire	new *type*
allocation et initialisation	new *type* (*liste-expr*)
allocation avec placement	new (*liste-expr*) *type*
allocation mémoire	new (*liste-expr*) *type* (*liste-expr*)
destruction	delete *pointeur*
destruction	delete[] *tableau*
conversion	(*type*) *expr*
pointeur sur membre	*objet* .* *pointeur*
pointeur sur membre	*pointeur* ->* *pointeur*

TAB. 3.1 – liste des opérateurs (*début*)

<table>
<tr><td>multiplication</td><td>expr * expr</td></tr>
<tr><td>division</td><td>expr / expr</td></tr>
<tr><td>reste</td><td>expr % expr</td></tr>
<tr><td>addition</td><td>expr + expr</td></tr>
<tr><td>soustraction</td><td>expr - expr</td></tr>
<tr><td>décalage à gauche</td><td>expr << expr</td></tr>
<tr><td>décalage à droite</td><td>expr >> expr</td></tr>
<tr><td>plus petit</td><td>expr < expr</td></tr>
<tr><td>plus petit ou égal</td><td>expr <= expr</td></tr>
<tr><td>plus petit</td><td>expr > expr</td></tr>
<tr><td>plus petit ou égal</td><td>expr >= expr</td></tr>
<tr><td>égal</td><td>expr == expr</td></tr>
<tr><td>différent</td><td>expr != expr</td></tr>
<tr><td>ET bit-à-bit</td><td>expr & expr</td></tr>
<tr><td>OU exclusif bit-à-bit</td><td>expr ^ expr</td></tr>
<tr><td>OU inclusif bit-à-bit</td><td>expr | expr</td></tr>
<tr><td>ET logique</td><td>expr && expr</td></tr>
<tr><td>OU logique</td><td>expr || expr</td></tr>
<tr><td>expression conditionelle</td><td>expr ? expr : expr</td></tr>
<tr><td>affectation</td><td>objet = expr</td></tr>
<tr><td>multiplication et affectation</td><td>objet *= expr</td></tr>
<tr><td>division et affectation</td><td>objet /= expr</td></tr>
<tr><td>reste et affectation</td><td>objet %= expr</td></tr>
<tr><td>addition et affectation</td><td>objet += expr</td></tr>
<tr><td>soustraction et affectation</td><td>objet -= expr</td></tr>
<tr><td>décalage à gauche et affectation</td><td>objet <<= expr</td></tr>
<tr><td>décalage à droite et affectation</td><td>objet >>= expr</td></tr>
<tr><td>ET et affectation</td><td>objet &= expr</td></tr>
<tr><td>OU inclusif et affectation</td><td>objet |= expr</td></tr>
<tr><td>OU exclusif et affectation</td><td>objet ^= expr</td></tr>
<tr><td>lancement d'exception</td><td>throw expr</td></tr>
<tr><td>séquence</td><td>expr , expr</td></tr>
</table>

TAB. 3.2 – liste des opérateurs (*suite*)

3.1.1 Opérateurs logiques

L'opérateur `&&` évalue d'abord son opérande gauche et, si le résultat est `false`, il rend `false` sans évaluer l'opérande droit, sinon il rend la valeur de ce dernier. De même l'opérateur `||` n'évalue son opérande droit que si le gauche vaut `false`. Ce comportement asymétrique fait que l'expression `i!=0 && 146%i==1` est correcte alors que `146%i==1 && i!=0` ne l'est pas (Quand `i` peut être nul !).

Les opérateurs de comparaison retournent `true` si la comparaison réussit, et `false` quand elle échoue.

Toute expression entière peut être utilisée pour un test, elle est convertie en `true` si elle est non nulle et `false` sinon.

Avant la norme de l'ISO (1995-1997) et donc sur les anciens compilateurs il n'existe pas de type booléen prédéfini, il en est de même pour les compilateurs **C**. On peut retrouver partiellement le bénéfice d'un type booléen en déclarant

```
typedef int bool;
enum { false, true};
```

Cette construction ne garantit pas que `false` et `true` sont les deux seules valeurs d'un `bool` mais elle permet d'écrire

```
bool b = i<3;
```

ce qui serait refusé si nous définissions `bool` comme un type distinct :

```
enum Bool {faux, vrai};
Bool b = (Bool)(i<3); //conversion explicite nécessaire
```

3.1.2 opérateurs d'affectations

C++ définit outre l'opérateur d'affectation simple `=`, les opérateurs `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=`, `^=`.

les opérateurs qui combinent une opération avec une affectation appliquent l'opération à leurs deux arguments et stockent le résultat dans l'argument gauche, qui doit être une référence à un objet.

Cette référence constitue la valeur de retour de l'expression, celle-ci peut donc être elle même utilisée comme opérande gauche d'une affectation.

Les deux expressions `(x+=x)*=2` et `x+=(x*=2)` sont correctes et donnent à la variable `x` les valeurs respectives $4*x$ et $3*x$ en notant x la valeur de `x` avant l'instruction.

Dans une affectation de nombres, l'opérande droit est converti au type de l'opérande gauche. Quand l'opérande gauche est un pointeur, l'opérande droit

doit être un pointeur admettant la conversion ou le littéral `0`. On ne peut affecter à un pointeur sur une constante qu'un pointeur sur une constante ou `0`. Le comportement des affectations pour les classes sera évoqué plus loin §7.6.2.

3.1.3 Opérateur virgule

L'opérateur virgule '`,`' évalue son opérande gauche, dont il abandonne la valeur, puis évalue son opérande droit qui fournit la valeur de l'expression. Le rôle de cet opérateur est donc de produire un effet de bord [1] durant l'évaluation du premier opérande.
Exemple :

```
int i,j,k;
for( i=1, j=1, k=1 ; i<100; i++,j++,k++){
.......
}
```

3.1.4 Expression conditionnelle

L'expression conditionnelle *cond?exp2 :exp3* est composée d'une condition et de deux expressions. Les expressions doivent être toutes deux d'un type arithmétique, ou d'un type pointeur, ou des objets d'une même classe. La condition est d'abord évaluée puis, si le résultat est différent de zéro, la seconde expression est évaluée sinon c'est la troisième. La valeur de la sous expression évaluée est la valeur de l'expression conditionelle .
Exemple :

```
char c1,c2;
...
c2=( c1>'a' && c1 <'z' || c1>'A' && c1<'Z')? c1 : '*';
```

3.2 Instructions

3.2.1 Instruction expression

Une expression suivie d'un «`;`» constitue une instruction. Les expressions les plus habituellement employées comme instruction sont les appels de fonctions et les affectations.

[1] voir définition en §3.4.

Instruction nulle

Un « `;` » constitue une instruction nulle qui n'a aucun effet. Nous nous en servirons comme corps d'itération et comme initialisation dans des boucles `for`.

3.2.2 Bloc d'instructions

Un bloc permet de composer plusieurs instructions en une seule *instruction bloc* résultante. Un bloc est composé d'une suite d'instructions placées entre accolades `{}`.

Nous utiliserons les blocs pour déterminer une nouvelle *portée* des variables locales à ce bloc, en particulier le corps d'une fonction est constitué d'un bloc.

Outre les fonctions la principale utilisation des blocs est le corps des boucles et les alternatives des instructions de sélection.

Bien que le langage autorise à mettre une simple instruction comme corps d'une structure de contrôle, cet usage provoque des erreurs aussi bien à la lecture du code que lors de sa modification. En particulier l'ajout à cette instruction isolée d'une seconde instruction contiguë casse la structure de contrôle. On adoptera donc la recommandation suivante :

Rec. 4.3 Toutes les primitives de contrôle de flux (`if`, `else`, `while`, `for`, `do`, `switch`, et `case`) doivent être suivies d'un bloc éventuellement vide.

3.2.3 Instructions de sélection

Instruction `if`

Elle est de l'une des formes :

```
if (expression) instruction
if (expression) instruction else instruction
```

L'expression doit être de type arithmétique ou pointeur : si elle est non nulle, la première instruction est évaluée, si elle est nulle l'instruction après le `else` est évaluée.

Pour les instructions `if` il est indispensable d'adopter une convention d'écriture et d'indentation des instructions de sélection de manière à repérer aisément les différentes alternatives. Quand elles incorporent des blocs d'instruction on écrira donc :

```
if (expression){
  instruction
    .....
```

```
  instruction
}else{
  instruction
    .....
  instruction
}//endif
```

On évitera les alternatives sans accolades comme

```
if (cond)
     inst1;
inst2;
inst3;
```

car il est fréquent qu'un programmeur distrait (ou habitué à *python* qui délimite ses blocs par des retraits) vienne ajouter :

```
if (cond)
     inst1;
     autre_instruction;
inst2;
inst3;
```

Le résultat est syntaxiquement légal, mais n'a pas la signification désirée.

Les instructions `if` se combinent pour donner des alternatives multiples, nous nous servirons du fait qu'une clause `else` se réfère toujours au `if` le plus proche pour les construire ainsi :

```
if (condition1){
   bloc1
}else if (condition2){
   bloc2
}else if (condition3){
   bloc3
.....
}else{
   bloc sinon
}//endif
```

Instruction `switch`

```
switch ( expression ){
  case expression constante: liste d'instructions
```

```
  ......
  case expression constante: liste d'instructions
  default : liste d'instructions
}
```

Si l'expression du `switch` a une valeur qui apparaît dans l'un des `case` qui suivent, alors les instructions de la liste correspondante, *ainsi que toutes les instructions suivantes* sont exécutées. L'instruction `break` permet de passer à la fin du `case` et est le plus souvent employée à la fin de chaque cas. Le cas `default`, optionnel mais recommandé, est exécuté quand aucune correspondance n'a été trouvée avec une des expressions constantes.
Exemple :

```
enum Jour {lundi, mardi, mercredi,
           jeudi, vendredi, samedi, dimanche}
Jour j;
....
switch (j){
case samedi:
  cout << "samedi"<<endl;
  break;
case dimanche:
  cout << "dimanche"<<endl;
  break;
case vendredi:
  cout << "vendredi: ";
default:
  cout "jour de travail"<<endl;
}
```

Dans l'exemple précédent les jours de la semaine provoqueront l'écriture de la chaîne «jour de travail», hormis vendredi qui écrira «vendredi : jour de travail». Ce type de programmation manque cependant de clarté.

Rec. 4.4 Les instructions qui suivent une étiquette `case` doivent se terminer par une instruction qui sort du `switch`.

Tous les cas doivent donc se terminer par `break` ou `return`. On écrira donc plutôt que l'exemple précédent :

```
switch (j){
  case samedi:
```

```
  {
    cout << "samedi"<<endl;
    break;
  }
  case dimanche:
  {
    cout << "dimanche"<<endl;
    break;
  }
  case vendredi:
  {
    cout << "vendredi: jour de travail"<<endl";
    break;
  }
  default:
  {
    cout "jour de travail"<<endl;
  }
}
```

Rec. 4.5 Tous les `switch` doivent avoir un `default`.

On écrira le cas par défaut même s'il semble ne jamais pouvoir être atteint, en effet on ne connaît pas l'évolution ultérieure du programme : un changement de type énuméré par exemple, peut faire apparaître de nouveaux cas.

3.2.4 Instructions d'itération

Les instructions d'itération, plus communément appelées **boucles** se présentent en **C**++ sous trois formes : le `while`, le `do` et le `for`.

Instruction `while`

`while(` *`expression`*`)` *`instruction`*

L'instruction est exécutée tant que l'expression est non nulle. L'expression est évaluée *avant* l'exécution de l'instruction.
Exemple :

```
int i(123),j(77);
while ( j%i ){
  j=j>i?j-i:i-j;
}
```

Instruction `do`

```
do instruction while( expression);
```

L'instruction est exécutée jusqu'a ce que l'expression *soit nulle*. L'expression est évaluée après l'exécution de l'instruction.
Exemple :

```
do { cout<<"OK"<<endl; } while( false);
```

écrit une fois `OK`.

La recommandation suivante est valable pour toute boucle :

Rec. 4.2 Modifiez la variable de boucle près de l'endroit où se trouve la condition de sortie.

Cette recommandation rend les boucles beaucoup plus lisibles, il est facile de trouver où est incrémenté le compteur de boucle. L'application de cette règle nous aidera aussi à choisir entre un `while` et un `do`.

Instruction `for`

```
for (initialisation; expression-test; expression-inc)
              instruction;
```

L'instruction *initialisation* est d'abord exécutée, puis *instruction* suivie de *expression-inc* sont évaluées tant que *expression-test* est non nulle. Si l'*initialisation* est une déclaration, d'après la norme ISO (*versions préliminaires avril 95, novembre 1997*) sa portée est limitée au bloc `for`, mais pour les compilateurs plus anciens elle s'étend jusqu'à la fin du bloc *englobant* le `for`.

Même dans le cas d'utilisation d'un ancien compilateur on appliquera la règle :

Règle 15.14 Ne réutilisez pas les variables déclarées dans une boucle `for`.

Cette règle implique aussi que l'on utilise toujours pour la boucle une variable locale.

Règle 4.1 Ne changez pas l'index de boucle à l'intérieur d'une boucle `for`.

Dans les cas où elle est utilisable la boucle `for` est supérieure aux autres formes de boucles car la condition de sortie et l'incrémentation de la variable de boucle sont groupées et clairement visibles. Si vous changez la variable dans le corps de la boucle vous trompez celui qui relit votre code : soit vous employez un mauvais type de boucle soit, plus souvent, vous n'exprimez pas correctement la condition de sortie.

À la place de :

```
int t[15];
int j;
for (int i(0); i<15; ++i){
  if ( T[i]==3){j=i; i=15;} //interdit
}
```

on écrira :

```
int i(0)
for (; i<15 && T[i]!=3 ; ++i){
}
```

Seule cette dernière forme nous permet de voir immédiatement que la condition de sortie de boucle est `i>=15 || T[i]==3`; notez que le *et* dissymétrique évite le dépassement de tableau.

3.2.5 Instructions de saut

Les instructions de saut peuvent provoquer une sortie de bloc, dans ce cas les variables automatiques allouées au moment de l'appel, et dont l'instruction de saut rompt la portée seront détruite par leur destructeur dans l'ordre inverse de leurs déclarations au moment du saut.

Instructions `continue` et `break` dans une boucle

L'instruction `continue` provoque la fin de l'itération courante de la plus petite boucle englobante, alors qu'un `break` exécuté dans une boucle la termine.
Exemple :

```
for (int i(0);;i++){
  if (i%2 == 0) continue;
  if  (i == 15) break;
  cout << i;
}
```

Instruction `return`

Voir §3.4.3

Instruction `go to` et instruction étiquetée

Un `go to` transfère le contrôle à l'instruction *de la même fonction* portant l'étiquette du `go to`.

La construction :

identificateur : instruction

ajoute une étiquette à l'instruction.

Ces deux instructions sont maintenant obsolètes.

Règle 4.6 Utilisez les `break` et `continue` à la place de `goto`.

Ceux-ci couvrent les principaux cas de sortie anticipée d'une boucle et il n'est nul besoin de recourir au `goto`. Un dernier cas de l'emploi de celui-ci était le traitement de certains anomalies rares, mais ici il a été remplacé par les exceptions.

3.2.6 Le choix d'une forme de boucle

Les trois formes de boucles présentées §3.2.4 sont trois formes équivalentes par leur puissance d'expression et bien entendu pour le code généré.

Pour tout problème donné il est possible au prix de quelques contorsions syntaxiques d'utiliser une de ces trois formes de boucle. Il se pose donc le problème du choix de la boucle.

Et bien entendu ce choix doit se faire en vue d'améliorer la lisibilité du code source. Pour toute boucle la question que nous nous posons est d'abord : « où est on sorti de la boucle, et pourquoi ? » ; cette question est celle de la condition d'arrêt. Une autre question est « Quelle propriété du programme reste vérifiée pendant tout le parcours de la boucle ? », le texte du programme ne peut à lui seul répondre à cette seconde question, mais il peut permettre de vérifier une assertion portée par un commentaire ou supposée par le lecteur du texte source.

La forme la plus générale de boucle, que l'on adoptera quand aucune autre forme plus particulière n'est adaptée, est celle d'un `while( true)` avec des `break`s pour sortir de la boucle, et des `continue`s pour sauter une itération.

Cette forme est adaptée aux boucles ou l'évaluation de la condition de sortie ne peut se faire ni en début ni en fin de boucle, ou bien quand de multiples conditions de sorties doivent être évaluées à des moments successifs. Cela peut être le cas de boucles interrogations comme celle du programme 3.1.

Il est bien sur possible avec une variable booléenne supplémentaire et en utilisant des `if` imbriqués de coder ce type de boucle avec un `while` à une seule condition de sortie. Vous êtes invités à essayer, et juger de vous même quelle forme vous semble la plus claire.

Programme 3.1 Boucle d'interrogation.

```
#include <iostream>
#include<string>
using string; using std::cout; using std::cin;
using std::endl;

int main(int argc, char * argv[]){
   string nom("toto");
   string nouvnom(nom);
   string reponse;
   const int maxessais(5);
   int nbessais(0);
   while (true) {
      if (nbessais++>maxessais) {
         cout << "maximum de "<< maxessais;
         cout << "essais dépassé"<<endl;
         break;
      }
      cout << "entrez un nom , Q pour garder le nom actuel: ¨";
      cout <<nom<<'"'<<endl;
      getline(cin,reponse);
      if (reponse=="Q") break;
      if (reponse=="") continue;
      nouvnom=reponse;
      cout << "conserver ¨"<<nouvnom<<"¨ [O/N/Q]"<<endl;
      std:: getline(cin,reponse);
      if (reponse=="Q"|| reponse=="O") break;
   }
   if (nbessais>maxessais || reponse == "Q") {
      cout << "nom inchangé: "<<nom<<endl;
   } else {
      nom=nouvnom;
      cout << "nouveau nom: "<<nom<<endl;
   }
  return 0;
}
```

Cette forme de boucle reste cependant la plus rare, nous n'avons habituellement pas besoin de cette généralité. Un cas plus particulier est celui ou la condition est unique et peut être testée en fin de boucle (mais ne peut pas l'être au début). C'est souvent le cas dans des boucles d'interrogation plus simples que la précédente.

Dans ce cas la forme `do{ ... } while (condition)` s'impose, mais elle ne représente encore qu'une faible proportion des boucles.

Le cas le plus usuel est celui ou la condition est unique et peut être testée en début de boucle. C'est le cas standard de l'utilisation du `while`. Ce cas comporte lui même un cas particulier où la condition d'arrêt porte sur une variable qui est augmentée en fin de boucle.

```
Tnum compteur(init);
while(compteur<limite){
  ...
  ++compteur
}
```

Une abréviation très claire pour ces boucles est la forme classique du `for` :

```
for(Tnum compteur(init);compteur<limite;++compteur){
  ...
}
```

Il convient de toujours préférer cette forme quand elle est applicable car elle condense dans sa première ligne les points clés de l'itération. La boucle `for` sera donc la forme la plus usuelle des boucles. Elle peut encore s'utiliser même quand les compteurs de boucles sont multiples et que la condition est complexe dès lors que la condition porte sur les compteurs, qu'elle est évaluable en début de boucle, et que les compteurs sont incrémentés en fin d'itération. Nous avons alors des boucles comme celle-ci :

```
for( int i(0),j(98),k(40);
     i<50 && j>i && k+j>3*i; ++i, j+=i/2,--k){
  ...
}
```

3.3 Exception

3.3.1 Les traitements d'erreurs

Considérons le petit programme suivant :

```
#include <iostream>
#include <string>
enum Jour {lundi,mardi,mercredi,jeudi,
           vendredi,samedi,dimanche};
const  string nomJour[]={"lundi","mardi","mercredi","jeudi",
                          "vendredi","samedi","dimanche"};
int main(){
   Jour j; //lundi
   j=Jour(2);
   cout << j <<" "<< nomJour[j] <<endl;
   j=Jour(10);
   cout << j <<" "<< nomJour[j] <<endl;
   j=Jour(500);
   cout << j <<" "<< nomJour[j] <<endl;
   return 0;
}
```

Certains compilateurs pourront nous avertir d'erreurs, d'autres ne le feront pas, mais à l'exécution ce programme va produire un accès à un élément hors du tableau qui produira une erreur.

Nous pouvons remplacer le tableau `nomJour` par une fonction qui teste son argument, par exemple :

```
string nomJour(Jour j){
   const  string nomJour[]={"lundi","mardi","mercredi","jeudi",
                            "vendredi","samedi","dimanche"};
   if (j<lundi || j>dimanche ) {
     exit 1;
   }
   return nomJour[j];
}
```

Mais l'appel à la primitive système `exit` va interrompre le programme englobant. Cette situation peut dans la phase de mise au point signaler clairement une erreur, mais à l'intérieur d'un logiciel important elle constitue une forme de réponse disproportionnée et dangereuse. Nous ne souhaitons pas que dans notre voiture, notre avion, notre machine-outil, moniteur cardiaque et même machine à laver, l'ordinateur se mette en grève quand on lui demande d'afficher un jour incorrect !

En production un `exit` ne convient pas. Nous pouvons alors utiliser la méthode : *signaler l'erreur et continuer*, c'est à dire :

```
string nomJour(Jour j){
   const  string nomJour[]={"lundi","mardi","mercredi","jeudi",
```

```
                              "vendredi","samedi","dimanche"};
   if (j<lundi || j>dimanche ) {
     return "erreur";
   }else{
     return nomJour[j];
   }
}
```

Bien sur malgré tous les avertissements que nous pouvons placer dans l'en-tête du programme, il sera bien improbable que l'utilisateur de la fonction vérifie qu'elle s'est déroulée sans erreur.

Par exemple le comportement standard de **C** est de retourner un pointeur zéro quand une demande de mémoire sur le *tas* ne peut être satisfaite. Comme ce cas empêchera sans aucun doute le programme de fonctionner, tout programmeur **C** doit savoir qu'il **faut** vérifier que l'allocation mémoire à réussi.

Est il besoin de dire ce qu'il en est dans beaucoup de programmes !

Nous sommes accoutumés à la stratégie du «cela ira bien, pour cette fois» et à ne traiter que les événements fréquents. Pour que les événements exceptionnels soient pris en compte il faut obliger leur traitement.

Mais, quel traitement prévoir ? Pour certaines utilisations l'abandon du programme après une erreur est approprié, alors que pour d'autres le retour d'une valeur spéciale permet de continuer le traitement. Souvent nous ne connaissons pas à l'avance les dans quel contexte nos fonctions seront utilisés, nous n'avons donc pas les éléments pour choisir.

3.3.2 L'utilisation d'une exception.

Les exceptions permettent d'indiquer l'erreur en laissant à l'utilisateur la possibilité de la traiter ou non, et le choix de la méthode pour traiter l'erreur.

Les exceptions sont des objets de classe, mais dans ce chapitre nous nous contenterons du cas particulier des structures, et même, pour commencer, d'une structure vide !

Lancement et nterception d'une exception

Le lancement d'une exception est fait par un ordre `throw` ; quand une exception est lancée, le déroulement du programme est altéré. Le contrôle quitte successivement tous les blocs de contrôles imbriqués depuis le plus intérieur où est lancée l'exception. Si il existe un bloc `try` où un gestionnaire de l'exception est prévu elle est interceptée et traitée ; sinon le programme est abandonné quand le contrôle atteint le bloc le plus externe du programme.

Programme 3.2 Exceptions pour les jours.

```
#include <iostream>
#include <string>
using std::cout; using std::cerr; using std::endl;
using std::string;
enum Jour {lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche};

struct MauvaisJour{};

string nomJour(Jour j){
   static const  string nomJour[]={"lundi","mardi","mercredi",
                        "jeudi","vendredi","samedi","dimanche"};
   if (j<lundi || j>dimanche ) {
      throw MauvaisJour();
   }
   return nomJour[j];
}

int main(){
   Jour j; //lundi
     try {
        j=Jour(2);
        cout << j <<" "<< nomJour(j) <<endl;
        j=Jour(10);
        cout << j <<" "<< nomJour(j) <<endl;
        j=Jour(500);
        cout << j <<" "<< nomJour(j) <<endl;
     }
     catch (MauvaisJour) {
        cerr << "Jour hors de  0..6" <<endl;
     }
     catch( ... ) {
        cerr << "exception inconnue" <<endl;
     }
```

Un bloc `try` est un bloc standard, avec les règles de portées habituelles, mais à la suite du bloc `try` des gestionnaires d'exceptions peuvent être spécifiés. Chaque gestionnaire est constitué d'un ordre `catch` suivi du type de l'exception traitée et d'un bloc qui comprend le traitement. La spécification `catch( ... )` intercepte toutes les exceptions. Dans l'exemple 3.2 le programme va écrire :

```
2 mercredi
Jour hors de  0..6
```

et se terminer.[2]

Quand une exception se produit dans une fonction nous pouvons parfois la traiter complètement, parfois nous n'avons pas les éléments pour le faire. Dans ce cas nous devons laisser l'exception interrompre la fonction pour permettre au code appelant de la traiter. Enfin il est parfois possible de traiter partiellement l'exception et de la déclencher à nouveau pour la signaler au programme appelant. Dans un bloc `catch` un ordre `throw` dépourvu de nom d'objet relance la même exception.

Le programme 3.3 présente une fonction qui imprime un nom de jour, les jours erronés sont interceptés et s'écrivent comme `"erreur"`, et l'exception est relancée.

Pour la première sortie le programme principal écrit sur la sortie standard `cout` :

```
133 erreur
```

puis le gestionnaire d'exception écrit `anomalie` sur la sortie erreur `err` :

Quand à la seconde sortie après avoir écrit :

```
500 erreur
```

L'exception ne trouvant pas de gestionnaire elle interrompt le programme.

3.3.3 Spécification d'exception

Il est souvent souhaitable de connaître les exceptions susceptibles d'être lancées par une fonction ou méthode. Nous pouvons documenter notre code avec une *spécification d'exception* qui est de la forme :

```
throw ( {Exception1},  {Exception2}, ... )
```

[2] `10` n'est pas écrit, parce que le déclenchement de l'exception interompt la sortie sur `cout` avant la fin de ligne.

Programme 3.3 Programme qui relance une exception

```
void printJour(Jour j){
   try{
      cout<< nomJour(j);
   }
   catch (MauvaisJour){
      cout << "erreur";
      throw;
   }
}

int main(int argc, char * argv[]){
   Jour j; //lundi
     try{
        j=Jour(133);
        cout << j <<" "; printJour(j);cout  <<endl;
     }
     catch (...) {
        cerr << "anomalie" <<endl;
     }
     j=Jour(500);
     cout << j <<" ";printJour(j);cout  <<endl;
     cout << "cette phrase n'est pas imprimée"<<endl;
     return 0;
}
```

Ici *Exception1*, ... représentent les classes des exceptions qui sont susceptibles d'être lancées par la fonction.

Certaines de ces exceptions peuvent être lancées directement par le code de la méthode, d'autres provenir indirectement des fonctions appelées. Le compilateur signalera une erreur si il détecte dans l'unité de traduction une exception qui ne figure pas dans la spécification d'exception et qui peut être lancée. Cette vérification n'est pas effectuée à l'édition de lien, donc il n'y a aucune garantie que la spécification d'exception soit complète.

Cette absence de vérification à l'édition de lien peut sembler un défaut, mais elle permet de rajouter à une fonction de bibliothèque une spécification d'exception sans recompiler *tous* les programmes qui utilisent la bibliothèque.

Si nous utilisons les spécifications d'exceptions l'en-tête de notre fonction sera :

```
void printJour(Jour j) throw ( MauvaisJour );
```

3.4 Fonction.

La notion de fonction répond à plusieurs motivations distinctes

- Le besoin d'avoir une action clairement identifiée, spécifiable et testable individuellement qui modifie son environnement. Cette notion reçoit dans d'autres langages le nom de procédure. Une telle modification de l'environnement par une fonction est appelée un **effet de bord** et doit être clairement indiquée. Les effets de bords sont sources de nombreuses erreurs en programmation classique. Les méthodes non constantes en programmation objet sont une forme policée de l'effet de bord puisque la modification est limitée à l'état de l'objet appelant.
- Une variante de la notion de procédure est une fonction qui regroupe simplement des instructions pour en permettre la compréhension, la vérification et le test. Une telle fonction provient généralement d'une analyse fonctionnelle descendante, et elle est préférée à un simple bloc car le bloc n'est pas une unité de test ou de vérification.
  Ces procédures correspondent à une bonne pratique de programmation si les effets de bords sont limités, et clairement documentés. Il est toujours préférable d'ajouter un paramètre à une fonction plutôt que d'accéder à une variable globale. De plus si le passage se fait par référence il n'implique aucun surcoût.
- Une fonction de programmation peut aussi simplement réaliser une fonction mathématique, c'est-à-dire calculer un résultat en fonction des arguments qui lui sont fournis et délivrer la valeur calculée, sans modifier ou même accéder à son environnement. De telles fonctions sont habituellement très sûres car elles peuvent être entièrement isolées et validées.

Rec. 4.7 Ne créez pas de fonctions trop complexes.

Puisqu'une fonction est la plus petite partie de code qui peut être vérifiée et testée indépendemment de son contexte elle doit rester simple.

Or la difficulté à comprendre et à tester une fonction dépend directement de sa complexité algorithmique, c'est-à-dire du nombre de structures de contrôle imbriquées. Il est donc indispensable de garder faible cette complexité.

Quand l'indentation laisse la moitié de l'écran vide, ce n'est pas la taille de l'écran qu'il faut améliorer, mais la conception du programme.

3.4.1 Identité et surcharge des fonctions

En **C++** une fonction est identifiée par son nom ainsi que par les nombres et types de ses arguments. Ce qui implique que deux fonctions différentes peuvent avoir le même nom si elles ont des arguments différents en nombre ou en type ; de telles fonctions sont dites *surchargées* . Pour identifier quel exemplaire d'une fonction surchargée est appelé il est nécessaire d'examiner le type de ses arguments effectifs et de tenir compte des conversions implicites qui ont lieu, c'est pourquoi il ne faut pas multiplier inutilement le nombre de ces conversions.
Exemple :

```
enum Couleur { blanc, jaune, rouge, vert, bleu};
enum Jour {dimanche, lundi, mardi, mercredi,
            jeudi, vendredi, samedi};
Couleur suiv (Couleur c){
    return ( c == bleu ? blanc : Couleur(c+1));
}
Jour suiv ( Jour j){
    return ( j == samedi ? dimanche : Jour(j+1));
}
....
Jour j(dimanche);
Couleur c(rouge);
j=suiv(j); // suiv(Jour)
j=suiv(bleu);// erreur pas de conversion de Couleur vers  Jour
```

Nous utiliserons les fonctions surchargées pour effectuer des tâches similaires avec des paramètres différents (voir règle 7.14 p. 133).

3.4.2 Passage des arguments

En **C**++ la sémantique du passage des arguments est *toujours* celle de l'initialisation. La valeur de l'argument effectif initialise l'argument formel.

Passage par valeur

C'est le type de passage d'argument le plus commun et le plus sûr.

Pour les types prédéfinis l'initialisation se fait par copie et donc le passage des arguments se fait par copie. Il en sera de même pour les objets de classe à ceci près que la classe définit elle-même la sémantique de la copie qui n'est pas obligatoirement une copie de ses membres (cf. §7.6.2).

Rec. 7.3 Passez les arguments de types prédéfinis par valeur à moins que la fonction ne doive les modifier.

Argument pointeur

Les arguments se comportent comme les autres arguments, c'est-à-dire que la valeur de l'identité passée en argument effectif est recopiée dans le pointeur constituant l'argument formel. Il est à noter que dans ce cas, il est possible de changer dans le corps de la fonction la valeur du pointeur argument formel, ainsi que la valeur de l'objet pointé, en provoquant un *effet de bord* qui peut être souhaité ou malencontreux.

Règle 7.8 Un argument pointeur ou référence doit toujours être déclaré `const` si la fonction ne change pas l'objet qui lui est lié.

Nous pouvons protéger l'objet référencé en déclarant `const` l'argument formel. Cette solution est imposée quand l'argument effectif est l'identité d'un objet déclaré constant puisque les règles de conversion implicite interdisent d'initialiser un pointeur ordinaire avec l'identité d'un objet constant.

On utilisera le passage par pointeur seulement dans le cas ou le passage par valeur et le passage par référence sont inapplicables.

Rec. 7.4 N'utilisez un argument de type pointeur que si la fonction ou une fonction qu'elle appelle stocke l'adresse du pointeur.

En **C** un cas fréquent de passage de pointeur est celui du passage d'un pointeur sur le premier élément d'un tableau. En **C**++ quand nous devons communiquer avec du code **C** nous déclarerons `const` un pointeur sur un tableau **C** quand le tableau est déclaré constant ou simplement quand la fonction ne le modifie pas cf. règle 7.8).

Un cas particulier des tableaux constant est celui des littéraux chaînes de caractères pour lesquels nous obtenons :

Règle 7.10 Utilisez seulement des pointeurs `const char *` pour accéder à des littéraux chaîne de caractères.

La manipulation de tableaux est donc peu sûre quand elle n'est pas protégée par un interface standard. Comme nous l'avons signalé au paragraphe §2.8.2 les tableaux restent un concept de bas niveau qui sera confié à des classes dites *concrètes* qui géreront la cohérence des données.

Nous n'auront donc généralement pas à employer des tableaux dans des programmes d'applications et nous éviteront surtout le passage d'arguments tableaux quand la taille du tableau peut varier d'un appel à l'autre.

Rec. 13.6 Utilisez une classe tableau au lieu des tableaux prédéfinis.

Les librairies de programmes offrent des patrons de *classes tableaux* qui permettent au programmeur d'applications d'encapsuler ces opérations. Nous aurons recours aux classes `vector` §10.1 de la STL (§10) pour traiter les tableaux, en particulier les tableaux de caractères seront remplacés par des *string* §11, *rope* ou *vector<char>*.

On utilisera toujours ces classes qui évitent de mêler la programmation de bas niveau aux programmes utilisateurs.

La section *Arguments de type tableau* §5.1 du chapitre §5 est donc principalement destinée au programmeur **C** et au concepteur de bibliothèques.

Argument de type référence

Un argument de fonction peut être une référence. Comme habituellement, le paramètre formel est initialisé avec le paramètre effectif à l'appel de la fonction. Dans le cas particulier d'une référence le paramètre formel reçoit l'identité du paramètre effectif et en devient un synonyme durant l'exécution de la fonction.

Exemple :

```
void triple (int & i) {i*=3;}
.........
int j(5);
triple(j); // j == 15
triple(2*j); // erreur
```

Les paramètres références constituent donc en **C**++ le moyen standard de modifier de l'intérieur d'une fonction des objets externes. Il est maladroit de se servir dans ce but des pointeurs comme il est d'usage de le faire en **C**.

Argument référence constante

Les références constantes permettent d'assurer que le paramètre réel n'est pas modifié par la fonction, elles économisent le coût de la copie de sa valeur. Notons que des arguments références constantes peuvent être employés partout où nous employons des valeurs et qu'ils offrent un niveau de sécurité identique ; ils sont donc particulièrement indiqués pour le passage des gros objets.

Initialisation d'un argument référence par une valeur Une référence à une constante peut aussi accepter en argument effectif le résultat de l'évaluation d'une expression, et elle identifie alors un objet temporaire créé pour la durée de l'appel, et initialisé avec l'expression.

En revanche il est impossible d'initialiser un argument de type référence non constante avec le résultat du calcul d'une expression.

Arguments par défaut

Certains arguments d'une fonction peuvent comporter des valeurs par défaut qui remplacent les arguments effectifs correspondants si ceux-ci manquent lors de l'appel. Les arguments par défaut doivent impérativement occuper la fin de la liste des arguments. Les arguments par défaut ne doivent être déclarés qu'une seule fois, et donc ne seront pas répétés lors de déclarations multiples. Il est donc recommandé de placer ceux-ci dans le fichier en-tête .

Règle 7.17 Placez les arguments par défaut dans la déclaration de fonction à l'intérieur de l'en-tête, pas avec la définition de fonction dans le fichier implémentation.

La valeur par défaut doit être connue au moment de l'appel de la fonction ; ce ne peut donc pas être une expression qui contient un autre argument.

En **C**$^{++}$ contrairement au **C** une fonction déclarée avec une liste vide d'arguments est une fonction sans argument. Une fonction avec une liste variable d'arguments serait indiquée en faisant suivre les arguments connus de points de suspension, mais ces fonctions qui ne peuvent être vérifiées à la compilation sont sources d'erreurs.

Rec. 13.4 Utilisez la surcharge de fonctions et les appels chaînés plutôt que des fonctions avec un nombre inconnu d'arguments.

3.4.3 Valeur de retour

Une fonction dont le type de retour n'est pas `void` doit retourner une valeur. En **C++** une fonction sans type de retour est prise par défaut comme retournant un entier.

L'instruction `return(`*`expression`*`)` *initialise* la valeur de retour avec l'expression spécifiée. Des conversions peuvent avoir lieu lors de cette initialisation.

Il faut se garder de retourner un pointeur ou une référence à une variable locale qui serait détruite avant qu'on ne puisse l'utiliser.

Règle 5.9 Une fonction ne doit jamais retourner un pointeur ou une référence à une variable locale en dehors de sa portée de déclaration, ni y donner accès par un autre moyen.

Exemple :

```
long triple(int i){
  return 3*i;       //conversion en long après multiplication.
}
long triple(int i){
  return 3*long(i); //conversion avant multiplication
}
int& triple(int i){
  return 3*i;       //erreur
}
int& triple(int i){
  int j(3*i);
  return j;         //erreur à l'exécution
}
void triple(int* i){
  *i*=3;            //OK mais préférer un référence
}
int& triple(int& i){
  return ( i*=3);   //OK
}
```

Le type de retour d'une fonction ne peut pas être un tableau, ni une fonction, mais ce peut être un pointeur sur une fonction ou un tableau. Quand le type de retour est complexe la déclaration sera rendue plus claire par un `typedef`.
Exemple :

```
typedef int  Ifunc(int);
Ifunc*  fpif(int);
```

Retour d'une référence

Le type de retour d'une fonction peut être aussi une référence. La fonction peut alors apparaître en partie gauche d'une affectation.
Exemple :

```
int & double(int& i){return i*=2;}
int k(3);
double(k)+=3;  // k==9
```

Cette caractéristique est utilisée pour pouvoir enchaîner des affectations et aussi pour extraire la place d'une donnée dans une structure complexe, comme l'illustre l'exemple suivant d'adressage dispersé.
Exemple :

```
struct T { int cle; float val;};
T tab[47];
const int vide=-1;      //cellule vide
T& cherche (int i){     //recherche par adressage dispersé
  const int dep(i%47);
  int k(dep), n(0);
  while ((tab[k].cle!=vide) && (tab[k].cle!=i)&& n< 47){
     k+=2*n++ + 1;
  }
  if (n>=47) { cerr << "plus de place" << endl; abort(); }
  tab[k].cle=i;
  return(tab[k]);
}
main(){
for (int i(0);i<47;++i) tab[i].cle=vide;
cherche(748).val = 1.5;
cherche(3456).val=cherche(748).val;
cout << cherche(3456).val <<endl;
}
```

Il est indispensable que l'objet dont on délivre l'identité existe dans l'environnement de l'appel de la fonction, il est donc impossible de rendre une référence à une variable locale à la fonction.
Exemple :

```
int& plusgrand(int i, int j){
  return i>j?i:j;  //erreur retour de la référence
                   // à une expression temporaire
}
int& plusgrand(int& i, int& j){
  if i>j{
    return i;
  }else{
    return j;    //OK retour de la référence à un objet externe
  }
  ......
int i(2), j(3), k(4);
plusgrand(plusgrand (i, j)+1,k)=0;      //erreur!
plusgrand( ++plusgrand (i , j), k)=0;  //qui change?
}
```

Chapitre 4

Organisation d'un programme

4.1 Division en fichiers sources

4.1.1 Fichiers en-tête

Le **C**++ admet la compilation séparée. Une unité de compilation peut utiliser des entités qui sont déclarées dans le fichier source mais définies dans une autre unité de compilation. Les références externes sont résolues par l'éditeur de liens. Pour s'assurer de la cohérence des déclarations d'objets ou fonctions utilisées dans plusieurs fichiers source, il est traditionnel d'employer des fichiers qui sont suffixés avec un `.h` en **C** pour **C**++ nous utiliserons `.hh`.

Style A.9 Les fichiers en-têtes seront suffixés par `.hh`

Règle 15.4 Les en-têtes fournies par le système doivent être placées entre des crochets `<...>`, les autres en-têtes entre guillemets `"..."`.

Les fichiers *en-têtes* sont inclus dans un autre fichier source par une instruction :

```
#include <fich.hh>
```

qui incorpore un fichier `fich.hh` *d'une librairie système*, ou bien :

```
#include "fich.hh"
```

qui incorpore un fichier utilisateur.

Rec. 15.5 Ne mettez pas de noms absolus de répertoires dans les directives `include`.

Nous éviterons de donner des chemins absolus dans les en-têtes pour ménager la portabilité, nous éviterons aussi les noms comportant des contre-obliques qui ne sont pas portables.

Pour faciliter la lisibilité des programmes comme pour diminuer les temps de recompilation nous choisirons de garder des fichiers de tailles modérées, généralement nous ne mettrons dans une unité de compilation qu'une classe ou un groupe de classes étroitement dépendantes.

Quand les classes proviennent de différents paquetages nous identifierons les fichiers qui implémentent un paquetage en utilisant un préfixe commun pour leurs noms.

Rec. 1.7 Groupez les fichiers apparentés en utilisant un préfixe commun.

Exemple :

```
#include "TPSDate.hh"  //Date du paquetage Gestion-Temps
#include "TPSTemps.hh" //Temps du paquetage Gestion-Temps
```

Nous mettrons dans un fichier en-tête :

- Les spécifications de classes.
- Les déclarations de fonctions.
- Les déclarations de données externes.
- Les définitions de constantes simples.
- Les `typedef` globaux
- Les énumérations.
- Les définitions de macro-instructions.
- les `include` d'autres fichiers en-têtes

Dans un fichier en-tête nous devons inclure les fichiers en-tête qui contiennent les définitions des classes utilisées dans le fichier courant. Dans le cas ou on utilise seulement un pointeur ou une référence il est inutile d'inclure l'en-tête de la classe. Une déclaration *anticipée* de la classe suffit, à condition de ne pas utiliser de méthode ou de déréférencer le pointeur.

On ne doit jamais supposer que d'autres fichiers en-tête sont déjà inclus. Des `#include` doivent figurer explicitement pour tous les fichiers nécessaires.
Exemple :

```
fichier Date.hh
#include<iostream>
   ....
fichier Personne.hh
#include "Date.hh"
  ...
```

```
fichier main.cc
#include "Personne.hh"
   ....
cout << "début" << endl; //sans erreur de compilation
                         //mais dépendance incorrecte
```

Règle 2.1 Chaque fichier en-tête doit être auto-suffisant.

Les fichiers en-têtes sont souvent imbriqués aussi il est utile avant d'incorporer un fichier de vérifier s'ils ne sont pas déjà inclus, on évite ainsi les inclusions multiples d'un même fichier, et surtout la boucle que provoquerait une inclusion récursive.

À l'entrée de tous les fichiers en-tête nous placerons une *garde* constituée d'une macro-instruction qui par sa définition évite les inclusions multiples.

Style A.7 La garde d'un `include` doit être le nom du fichier en-tête avec tout les caractères illégaux remplacés par des caractères soulignés.

```
#ifndef DATE_HH
#define DATE_HH
....
en-tête
.....
#endif
```

4.1.2 Fichier des fonctions en-lignes

Les fonctions en-lignes ne peuvent être compilées séparément elles doivent donc être placées à la suite de l'en-tête.

Rec. 2.4 Les définitions des fonctions en-lignes seront placées dans un fichier séparé.

Style A.10 Les fichiers de définition `inline` doivent avoir l'extension `.icc`

Pour la mise au point il est souvent préférable de supprimer l'expansion en-ligne ce que nous effectuerons avec une macro-instruction.
Exemple :

```
fichier Date.hh
    ......
   déclarations
     .......
#ifndef NO_INLINE
#include "Date.icc"
#endif

fichier Date.icc
#include.... inclusion pour les fonctions inline
#ifdef NO_INLINE
#define inline
#endif
  ........
  définition des inline
  ........
#ifdef NO_INLINE
#undef inline
#endif

fichier Date.cc
    ........
   définition des fonctions
    ........
#ifdef NO_INLINE
#include "Date.icc"
#endif
```

Quand nous incluons des fichiers `inline` nous devons aussi inclure les en-têtes qui sont nécessaires au corps des fonctions `inline`, sans être demandées pour la déclaration de ces fonctions [1]. Nous inclurons ces en-têtes au début du fichier `.icc`

4.1.3 Inclusion des fonctions patrons

Les fonctions patrons sont traitées différemment des autres fonctions, même si elles ne sont pas `inline` elles sont d'abord seulement précompilées, la génération de code n'a lieu qu'à l'instanciation du patron. Elles seront placées séparément.

[1] Celles qui sont nécessaires à la déclaration des fonctions ont déjà été inclues dans le `.hh`

Rec. 2.5 La définition des fonctions patrons d'une classe doit être placée dans un fichier séparé.

La manière de traiter les patrons diffère d'un compilateur à l'autre, certains demandent de les placer dans l'en-tête d'autres dans un répertoire spécial,...

La solution est de créer un fichier pour les patrons et de faire des inclusions suivant les demande du compilateur utilisé.

Nous adopterons un suffixe `.tcc` pour les définitions des fonctions et méthodes patrons.

Rec. 15.15 Une seule directive `include` est nécessaire pour les `template`.

4.1.4 Fichiers sources et portée

On désigne par *unité de compilation* l'ensemble des fichiers sources qui par le jeu des `#include` sont compilés en une fois.

Une unité de compilation définit en **C**++ la portée des variables globales `static` et `const`, la portée des fonctions non membre `static` et `inline`, ainsi que celle des synonymes `typedef`.

Les autres variables globales sont partagées par les fichier qui sont *liés* ensemble si leur portée n'est pas contrôlée par un espace de nom (voir §2.1.6). Exemple :

```
// fichier fich1.cc
int a(1);
int b(1)
extern int c;
int f(int,int);       //une première fonction f
int g(){c=f(a,a);...} // modifie c dans fich2

//fichier fich2.cc
int c(4);
extern int a;
extern int b;
float f(float,float){....} //une seconde fonction f
int g();              //fonction définie dans fich1
void main(){
  cout <<a<<endl;   // a défini dans fich1
  ....
  int a(3);
  ....
```

```
  b=f(a,::a);      // b,f,::a dans fich1; a local
}
```

Il est important de différencier les variable *locales* au `main` des variables globales dont la portée dépasse l'unité de compilation (à l'exception des variables statiques). Ainsi les déclarations suivantes sont illégales :

```
// fich1.cc
int a(2);
int g(int i){return i;}

// fich2.cc
int a(3);
int g(int i){return i;}  // deux définitions
```

4.2 Documentation des programmes

La documentation des programmes est un élément essentiel d'un logiciel. Elle doit principalement permettre de montrer qu'un logiciel est conforme à ses spécifications. Pour les données il s'agit de montrer la relation entre l'état concret constitué par ces données et l'état abstrait défini pendant la conception. Pour les fonctions, les commentaires doivent permettre de vérifier que le programme est conforme à ses spécifications. Si les spécifications des fonctions ne sont pas contenues dans un autre document, elles doivent aussi accompagner leur déclaration.

On devra se rappeler que les commentaires seront utilisés par d'autres programmeurs pendant des phases de relecture, de test et de maintenance du programme. Il est donc important de limiter au minimum la part d'implicite dans ces commentaires. Les habitudes de codage changent d'un individu à un autre et doivent être explicitement rappelées il est souhaitable qu'elles fassent l'objet d'une norme documentée comme celle qui accompagne ce cours (*cf.* annexe §A).

La documentation d'un programme se fait souvent en ajoutant des commentaires au code. C'est insuffisant quand le projet compte plus que quelque centaines de lignes.

La programmation littérale initiée par le langage *WEB*[2] combine code et documentation en un même fichier dont sont extraits d'une part le code du programme et de l'autre sa documentation.

Une variante très utilisée de cette solution, est de placer la documentation dans des blocs spéciaux de commentaires, elle peut ensuite être extraite grâce

[2]Synonyme, mais sans rapport avec la *toile* internet

à un programme qui analyse ces commentaires et génère la documentation sous forme texte ou hyper-texte.

Cette méthode à été systématisée en java avec le paquetage `javadoc`, et est aussi très utilisée en Php, Python, C, **C**++ .

Pour ce dernier les programmes de documentation les plus utilisés sont `doxygene` et `doc++` mais aussi `ccdoc`, `scandoc`, ...

4.2.1 L'identification du fichier

Rec. 3.1 Tout fichier doit contenir un commentaire de copyright.

Rec. 3.2 Tout fichier doit contenir un commentaire avec une description du contenu du fichier.

Rec. 3.3 Tout fichier doit contenir une chaîne constante locale qui identifie le fichier.

Les systèmes de gestion de versions offrent un interface qui permet aisément de gérer ces champs. Ils peuvent être générés automatiquement par les lots clés des système RCS ou CVS, qui documentent aussi l'évolution du programme.

4.2.2 La documentation de l'algorithme.

Tous les commentaires commenceront par une double barre oblique `//`. Les commentaires dans le style du langage **C** sont à rejeter en **C**++ car puisqu'il ne peuvent pas s'imbriquer, l'ajout d'un commentaire provoque une erreur quand le champ délimité recoupe un commentaire précédant.

Rec. 3.4 Utilisez `//` pour les commentaires.

La recommandation 1.2 et la discussion de la page 20 s'applique aussi aux commentaires :

Rec. 3.5 Tous les commentaires seront écrits en anglais.

Les commentaires seront séparés en deux groupes :
- L'identification du fichier.
- La documentation de l'algorithme.

Voici une liste de conseils pour l'écriture des commentaires :

1. Ce qui peut être exprimé par le langage ne fait pas l'objet de commentaire. Ainsi on écrira :

```
const int marignan(1515);
```

et non :

```
int marignan(1515); //attention marignan doit être constant
```

ni, non plus :

```
const int marignan(1515); //entier constant, qui vaut 1515
```

2. Pour chaque fichier on indiquera le contenu du fichier et les dépendances avec les autres fichiers.
3. Pour chaque classe on donnera un commentaire exprimant la manière de représenter l'état abstrait d'un objet.
4. Pour chaque méthode la pré-condition et la post-condition doivent être exprimées par une assertion ou par un commentaire.
5. La signification d'une variable globale doit être indiquée, ainsi que les fonctions qui l'utilisent et celles qui la modifient.
6. L'équivalent abstrait des alternatives complexes sera indiqué : il est utile en particulier de documenter les `else` des alternatives à branches multiples.
7. Pour chaque boucle complexe on indiquera l'*invariant* de la boucle, c'est à dire une condition qui est vérifiée initialement et à chaque passage dans la boucle. Cette condition est donc vérifiée à la sortie de la boucle. Les seules condition que l'on peut supposer réalisées à la sortie d'une boucle sont la condition d'arrêt (i.e. la négation de la condition pour continuer) et l'invariant.
8. Pour prouver la terminaison de chaque boucle on devrait donner une expression entière qui décroît strictement à chaque passage dans la boucle. On pourra cependant s'en dispenser dans les cas où une telle expression est *clairement visibles* c'est à dire ceux où, dans une boucle, une variable entière croît strictement en restant inférieure à une borne donnée, ou décroît strictement en restant supérieure à une borne. Cette expression monotone prouve que la boucle se termine.

4.3 Le préprocesseur

Le préprocesseur est utilisé dans une phase qui *précède* la compilation. Il sert principalement :
- À inclure des fichiers externes
- À inclure, ou exclure des parties du code suivant des conditions prédéfinies.
- À définir des *macro-instructions* qui seront expansées dans le texte avant compilation.

4.3.1 Les macro-instructions

Syntaxe des directives du préprocesseur

Les directives du préprocesseur sont constituées de lignes qui commencent par le caractère # éventuellement précédé d'espaces.

Chaque directive est constituée par une ligne ou plusieurs lignes raccordées par le caractère d'échappement \ situé en fin de ligne.

Définition des macro-instructions simples

La ligne :

```
#define identificateur suite de lexèmes
```

demande au préprocesseur de remplacer l'identificateur par la suite de lexèmes dans tout le texte restant du programme.

Après remplacement le texte sera de nouveau examiné pour effectuer de nouvelles substitutions.

Une macro-instruction ne peut pas être définie plusieurs fois. (Seules des définitions strictement identiques sont autorisées.)

On peut supprimer une définition de macro-instruction avec :

```
#undef identificateur
```

Style A.6 Le nom des macro-instructions est entièrement en majuscules.

Les noms des macro-instructions sont les seuls noms qui seront écrits entièrement en majuscules, cela évite d'employer par mégarde un nom de macro-instruction pour une variable, fonction ou classe, qui ne sont pas soumis aux mêmes règles de syntaxes.

À coté des macros instructions simples il existe aussi des macros instructions qui admettent des arguments. Elles sont très utilisées en **C** où elles pallient les insuffisances du langage. En **C**++ les fonctions en-lignes et les patrons rendent leur utilisation rare. La descriptions des macros-instructions avec arguments est donnée §5.2.1.

Utilisation des macro-instructions

Comme décrit §4.1.1 les macros instructions sont principalement utilisées en **C**++ pour l'inclusion conditionnelle des fichiers, ainsi que pour positionner des options de débogage. Elles seront aussi utilisées en collaboration avec les *makefile* pour la gestion de configuration.

En **C** les macros sont utilisées pour les définitions de constantes, les définitions de type et les fonctions en-lignes. Cet usage est à proscrire en **C**++ qui incorpore ces concepts dans le langage même car elle court-circuite le contrôle de type.

Règle 13.5 N'utilisez pas de macro-instructions à la place des constantes, `enum`s et `typedef`s.

On utilisera en revanche les macros-instructions pour définir les mots clés non implémentés par le compilateur utilisé et éviter de modifier le code pour s'adapter à un produit qui n'est pas aux normes actuelles du langage.
Exemple :

```
#ifdef NO_BOOL;
#define int bool
enum {false,true};
#endif
#ifdef NO_EXPLICIT
#define explicit
#endif
```

Rec. 15.13 Utilisez des macro-instruction pour détourner l'utilisation des mots-clés non implémentés.

Une autre utilisation traditionnelle des macros-instructions était la génération de fonctions similaires qui ne différaient que par le type de leurs arguments. Les fonctions patrons 9.2 ont maintenant rendus cette utilisation obsolète.

Nous pouvons donc constater que l'évolution du langage **C**++ à cantonné le préprocesseur à son rôle de gestion de configuration, et que l'on doit maintenant éviter de l'employer pour simuler des constructions absentes du langage.

4.3.2 L'inclusion de fichier

Une ligne de la la forme :

```
#include <nom de fichier>
```

provoque l'inclusion du fichier de bibliothque *nom de fichier* qui sera recherché dans des répertoires dépendant de l'implémentation.

Une ligne de la la forme :

```
#include "nom de fichier"
```

provoque l'inclusion du fichier d'en-tête *nom de fichier* qui sera d'abord recherché dans le répertoire contenant le fichier source, puis s'il n'est pas trouvé, dans les mêmes répertoires que précédemment.

Une ligne de la la forme :

```
#include suite de lexèmes
```

sera interprétée en substituant les macros instructions présentes dans les lexèmes. Le résultat des substitutions doit correspondre alors à une des deux formes précédentes et le texte résultat sera alors utilisé comme ci-dessus.

Quand un fichier est inclus par un `#include` son contenu est lui-même traité par le préprocesseur, il peut en particulier y avoir des inclusions imbriquées.

L'utilisation des include est décrite en §4.1.1.

4.3.3 Compilation conditionnelle

Une compilation conditionnelle a la forme suivante :

```
#ligne condition
    partie {si}
# elif expression constante
    partie {sinon-si}
# elif expression constante
     .......
# else
    partie {sinon}
# endif
```

La *ligne condition* peut avoir les trois formes suivantes :

```
# if expression constante
# ifdef identificateur
# ifndef identificateur
```

La première forme teste si l'expression constante est non nulle. L'expression constante peut contenir l'opérateur :

```
defined ( identificateur)
```

ou

`defined` *`identificateur`*

qui est remplacé par 1 si l'identificateur est défini (c'est-à-dire à été l'objet d'un `#define`) et par 0 sinon.

L'instruction `#ifdef` est une abréviation de `#if defined` et de même `#ifndef` est une abréviation de `#if ! defined`.

Quand le macro-préprocesseur rencontre une instruction `#if` il évalue en séquence les conditions. La partie de programme se trouvant sous la première condition non nulle est incluse dans le programme alors que les autres sont effacées. Si toutes les conditions sont nulles et qu'il y a une *partie sinon* celle-ci est incluse.

Les expressions constantes figurant dans les conditions font l'objet des substitutions de macro-instruction avant évaluation.

Les expressions constantes ne peuvent pas contenir de `const int`, d'énumérations ni l'opérateur `sizeof`.

4.3.4 Le contrôle de la compilation

Liste des macro-instructions prédéfinies

`__LINE__` est le numéro de ligne du fichier source.

`__FILE__` est le nom du fichier source.

`__DATE__` et `__TIME__` la date et l'heure de la traduction du fichier source.

`__STDC__` défini par l'implémentation (*indique le langage standard*).

`__cplusplus` défini quand on compile du **C++**.

Nom des fonctions.

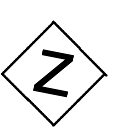
Le compilateur GNU CC définit des identificateurs représentant le nom de la fonction courante. *Ce ne sont pas des macros-instructions*, ni des variables ; mais des littéraux définis par le compilateur qui sont employés avec les macros `__LINE__` et `__FILE__` pour composer les messages d'erreurs.

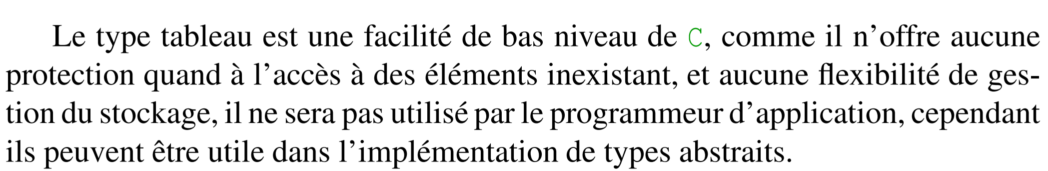
L'identificateur `__FUNCTION__` (`__func__` en **C**) contient le nom de la fonction englobante la plus proche de l'endroit ou il apparaît dans le texte source. L'identificateur `__PRETTY_FUNCTION__` contient une représentation du prototype de la fonction.

Chapitre 5

Quelques points plus techniques

5.1 Arguments de type tableau





Le type tableau est une facilité de bas niveau de `C`, comme il n'offre aucune protection quand à l'accès à des éléments inexistant, et aucune flexibilité de gestion du stockage, il ne sera pas utilisé par le programmeur d'application, cependant ils peuvent être utile dans l'implémentation de types abstraits.

Puisque les noms de tableaux s'évaluent comme l'identité du premier élément, passer en argument un tableau d'objets de type `T` à une fonction, c'est passer une valeur de type pointeur sur `T`, c'est à dire `T*` .

Les éléments du tableau ne sont donc pas passés par valeur, contrairement aux arguments scalaires. Quand les valeurs d'un tableau ne doivent pas être modifiées il faut déclarer l'argument comme `const T *`, et `T*` seulement si la fonction modifie les éléments du tableau.
Exemple :

```
void f( int* v){
   v[5]=0;
}
int g( const int * v)
   return v[5];
}
int t[]={1,2,3,4}
int u[]={5,6,7,8,9,10};
f(u); cout<<g(u)<<endl;  // 0
```

Comme la structure d'exécution du programme ignore sur quel objet pointe une variable pointeur, rien ne peut garantir à la compilation qu'une fonction arbitraire qui accède à la mémoire par un pointeur sur un type non constant ne modifie pas une zone à laquelle elle ne devrait pas avoir accès.

```
f(t);   // Détruit probablement une partie de u
```

5.1.1 Argument tableau de taille variable

Un argument formel tableau peut correspondre à des arguments effectifs dont les tailles seront différentes d'un appel à l'autre. Un tel argument est parfois abusivement appelé *tableau variable*. Il est alors nécessaire de déterminer la taille du tableau à l'intérieur de la fonction appelée. Mais la taille du tableau n'est pas enregistrée dans la structure mémoire qui le contient, et si le compilateur peut donner la taille de toute structure mémoire connue avec l'opérateur `sizeof` il est impossible de connaître la taille d'un argument qui ne peut être déterminé de manière statique.

Il appartient au programmeur d'adopter et de respecter une convention qui permet d'accompagner le pointeur sur le premier élément du tableau de sa taille. Cette convention est mise en œvre en **C**++ grâce à l'encapsulation qui permet de cantonner les manipulations du tableau dans une bibliothèque.

Deux conventions sont utilisées pour marquer la fin d'un tableau.

- La première est d'utiliser une *sentinelle* ou *drapeau*, c'est à dire une valeur spéciale du même type que les éléments du tableau, mais qui ne peut être employée que pour marquer la fin du tableau. Cela implique que l'on puisse trouver une telle valeur dans le type des éléments du tableau , ou bien représenter les éléments dans un type plus étendu. Cette solution est communément utilisée pour les chaînes de caractères où la valeur '\0' sert de sentinelle.
- Une seconde solution est de fournir à la procédure en même temps qu'un pointeur sur le premier élément du tableau, la taille de celui-ci.

5.1.2 Cas des tableaux multidimensionnels

Les tableaux à plusieurs dimensions sont plus difficiles à manipuler. Pour le tableau `t` suivant :

```
float t[4][3];
```

`t[0]`..`t[3]` sont quatre pointeurs sur des tableaux de trois `float`s. Nous considérons habituellement `t` comme une matrice stockée ligne par ligne. Une fonction qui reçoit en argument un tel tableau de *taille fixe* peut l'écrire de manière simple comme suit :

```
void ecriretab(float v[4][3]){
    for ( int i (0); i<4;i++){
```

```
        for (int j (0): j<3; j++){
            cout << ' ' << v[i][j];
        }
        cout << endl;
    }
}
```

Chaque pointeur `v[i]` est du type `float [3]` qui correspond à une ligne. Nous pouvons donc traiter un tableau dont le nombre de lignes est inconnu par le programme :

```
typedef float ligne[3];
void ecriretab(ligne* v, int dim1){
    for ( int i(0); i<dim1;i++){
        for (int j(0); j<3; j++){
            cout << ' ' << v[i][j];
        }
        cout << endl;
    }
 }
```

L'en-tête de ce programme pourrait aussi être écrite :

```
void ecriretab(float v[][3], int dim1)
```

Maintenant, si nous ne disposons plus d'une *constante* qui donne le nombre d'éléments par ligne, nous ne pouvons plus formuler exactement le type de `*v`. La solution est de considérer le tableau de `dim1` lignes dont chacune comporte `dim2` éléments comme un tableau de `$dim1 * dim2$` éléments. Comme le nom du tableau se convertit en `float**`, il nous faut, pour le considérer comme un tableau à une seule dimension, le convertir explicitement en `float*`, soit :

```
void ecriretab(float** v, int dim1, int dim2){
    float* v1=(float*) v;
    for ( int i (0); i<dim1;i++){
        for (int j (0); j<dim2; j++){
            cout << ' ' << v[i*dim2+j];
        }
        cout << endl;
    }
 }
```

Mais il est exceptionnel que l'on doive utiliser un tel code en C++ , il sera presque toujours remplacé par l'utilisation de vecteurs.



5.2 Le préprocesseur (compléments)

5.2.1 Définition des macros-instructions avec arguments

La ligne :

```
#define identificateur(liste d'arguments) suite de lexèmes
```

définit une macro-instruction avec arguments.

Dans la suite du texte toute apparition d'une forme :

```
identificateur(arg1,arg2  ....)
```

où chaque argument est constitué de lexèmes ne comprenant pas de caractère virgule (sauf entre guillemets ou à l'intérieur de parenthèses), est considérée comme un appel de la macro.

Le macro-préprocesseur substitue alors dans le corps de la macro les arguments effectifs aux arguments formels. Il s'agit d'un remplacement purement textuel, Les arguments sont eux-même examinés et leurs macros remplacées *avant la substitution*.

Le texte résultant est ensuite soumis à de nouvelles substitutions. Cependant une fois qu'une macro-instruction a été remplacée une fois l'identificateur perd son caractère de macro dans le texte de remplacement. Ce qui fait qu'il n'y a pas de remplacement récursif.

Après le remplacement des macros les chaînes de caractères adjacentes sont concaténées.

L'emploi des parenthèses dans les macros permet d'éviter l'application de règles de priorité malencontreuses dans les expressions arithmétiques.

Par exemple on définira le maximum de deux nombres par :

```
#define max(a,b) ( (a) > (b) ? (a) : (b))
```

où les parenthèses sont indispensables pour assurer que les expressions seront correctement évaluées.

L'exemple suivant illustre les remplacements successifs :

```
#define x 1+0
#define f(a) f(x*a)
#define g(a) f((x)*(a))
f(1+0)
#undef x
#define x 0+1
g(1+1)
```

il produit :

```
f(1+0*1+0)
f(0+1*(0+1)*(1+1))
```

5.2.2 L'opérateur

Quand une substitution est précédée de #, l'argument est entouré de guillemets avant la substitution. Les «"» inclus dans l'argument sont protégés par un \.
Exemple :

```
#define affiche(x) cout << "valeur de " << #x << x <<endl
affiche(toto)
```

s'expanse en :

```
cout << "valeur de " <<"toto" << toto<<endl;
```

5.2.3 L'opérateur

Quand après un remplacement de paramètres deux lexèmes sont séparés par ## et *avant de réexaminer* le texte, les deux lexèmes séparés par ## sont concaténés.
Exemple :

```
#define colle(a, b) a ## b
#define xcolle (a , b) colle ( a, b)
#define SALUT "bonjour"
#define UT UT " à tous"
colle( SAL, UT)
xcolle( SAL, UT)
```

s'expanse en `"bonjour"` et en `"bonjour" "à tous"`, ce dernier sera finalement fondu en une seule chaîne `"bonjour à tous"`.

5.2.4 Contrôle des lignes

Une directive du préprocesseur de la forme :

```
#line suite de chiffres
```

permet de renuméroter les lignes qui suivent à partir du nombre donné en représentation décimale.

La directive :

```
#line suite de chiffres "suite de caractères"
```

renumérote les lignes et change le nom présumé du fichier source.

5.2.5 Directive d'erreur

Les erreurs détectées à la compilation pourront être signalées par :

```
#error suite de lexèmes
```

5.2.6 Directive «pragma»

La directive :

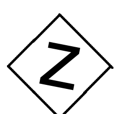
```
# pragma lexème
```

Donne une indication de compilation, un «pragma» qui n'est pas reconnu par le compilateur est ignoré.

Rec. 15.8 N'utilisez pas les pragmas.

Il est préférable d'éviter ces instructions destinées au compilateur dans le texte du programme. Si elles sont vraiment nécessaires elles pourront être incluse lors de la configuration, avant la compilation.

5.3 Unions



Une *union* est une structure dont tous les membres sont implantés à la même adresse et se recouvrent. Un objet de type union comprend donc un champ de l'un des types utilisés dans l'union. Pour pouvoir exploiter cette valeur il est nécessaire de connaître son type. Seul celui-ci fournit la clé qui permet de décoder la valeur stockée par la machine.

C++ ne fournit pas de procédure qui permette d'assurer qu'une valeur est lue avec le type correct : cette tâche revient à l'utilisateur. Plusieurs cas sont à distinguer :

– L'utilisation des unions pour faire du transtypage :
  C'est une utilisation commune dans les langages offrant peu ou pas de moyens pour effectuer des conversions. On entre une valeur d'un type et on «trompe» le compilateur en la lisant comme d'un autre type. Il est évident que le code ainsi obtenu dépend de la machine, de la version du système, et du compilateur employés, et est donc très peu fiable. Elle est absolument à proscrire en **C**++ .
– L'emploi d'une étiquette qui indique le type de la donnée. C'est la méthode la plus fiable d'emploi des unions, l'accès doit être contrôlé par une classe qui englobe l'union et assure qu'on ne peut accéder à la donnée avec un type erroné. Mais dehors de classes concrètes pour qui l'économie d'espace et de temps est critique on préfèrera l'héritage et la liaison dynamique.

– L'emploi d'une fonction qui par le contexte, permet de choisir le type correct : cette solution est une extension de la précédente. Elle doit de même être protégée par une classe qui fournit l'interface. A cause de la complication supplémentaire elle est plus dificile à déboguer, car il est plus simple de vérifier une étiquette qu'une fonction générale. On ne l'emploiera qu' exceptionnellement quand la place est si critique que les quelques positions binaires de l'étiquette sont encore trop pénalisantes.

Voici un exemple pour une utilisation sure de l'union dans les quelques cas exceptionnels ou elle est nécessaire.

```
class Ent_Cars4 {
public:
  enum Etiq (entier, cars);
  Ent_Cars4(long e): etiq( entier), i(e) {}
  Ent_Cars4(char* Chaîne):
    etiq(cars){strncpy(c,Chaîne,4)};
  Etiq sorte{return etiq;}
  int entval();
  // rend l'entier 0 si ce sont des caractères
  void copier(char* Chaîne);
  //copie les caractères Chaîne contient 4 cars+\0
  // 0  si entier
private:
  Etiq etiq;
  union {
    char c[4];
    long i;
  }
};

void Ent_Cars4::copier(char* Chaîne){
  if (etiq==cars) strncpy(Chaîne,c,5);
  else strncpy(Chaîne,"",5);
}

inline int entval(){
  return (etiq=entier)?i,0;
}
```

Mais sauf dans ces cas exceptionnels on s'abstiendra d'utiliser des unions, leur emploi révèle généralement un défaut de conception.

Rec. 13.7 N'utilisez pas d'`union`s.



5.4 Utilités de la bibliothèque standard C.

5.4.1 Les fichiers de limites.

Les valeurs des tailles des types prédéfinis sont des constantes définies dans des fichiers `include`.

Le fichier `<limits.h >` définit les constantes suivantes :

< limits.h >	
CHAR_BIT	nombre de bits par caractère
CHAR_MIN	valeur minimale d'un `char`
CHAR_MAX	valeur maximale d'un `char`
UCHAR_MIN	valeur minimale d'un `unsigned char`
UCHAR_MAX	valeur maximale d'un `unsighed char`
SCHAR_MIN	valeur minimale d'un `signed char`
SCHAR_MAX	valeur maximale d'un `signed char`
SHRT_MIN	valeur minimale d'un `short int`
SHRT_MAX	valeur maximale d'un `short int`
INT_MIN	valeur minimale d'un `int`
INT_MAX	valeur maximale d'un `int`
LONG_MIN	valeur minimale d'un `long int`
LONG_MAX	valeur maximale d'un `long int`
USHRT_MAX	valeur maximale d'un `unsigned short`
UINT_MAX	valeur maximale d'un `unsigned int`
ULONG_MAX	valeur maximale d'un `unsigned long`

Le fichier `<values.h >` quant à lui définit (entre autres) les valeurs :

< values.h >	
MINFLOAT	valeur minimale d'un `float`
MAXFLOAT	valeur maximale d'un `float`
MINDOUBLE	valeur minimale d'un `double`
MAXDOUBLE	valeur maximale d'un `double`

Chapitre 6

Le paquetage d'entrées sorties

Les entrées sorties ne font pas partie intégrante du langage **C++** mais sont fournies sous la forme d'un paquetage standard. L'interface de ce paquetage figure dans <iostream>.

Cet interface est maintenant placé dans l'espace de nom std.

Pour ne pas alourdir l'écriture nous supposerons que l'espace std a été rendu accessible par une clause :

```
using namespace std;
```

et nous utiliserons dans ce chapitre les noms de cet espace sans les préfixer systématiquement par un std::.

Les entrées sorties sont représentées en terme de flots de caractères. Un **flot de caractères** provient de (istream) ou se dirige (ostream) vers l'environnement extérieur.

Les primitives de sortie fournies par le paquetage convertissent un objet de type prédéfini en suite de caractères qui sont déposés sur un flot de sortie, alors que les primitives d'entrée extraient les caractères d'un flot d'entrée et les stockent après conversion dans un objet en mémoire. Ces primitives sont extensibles par surcharge : les utilisateurs ont la possibilité de redéfinir ces opérations de manière à ce qu'elles puissent aussi s'appliquer aux objets de leurs classes.

Règle 13.2 Utilisez la bibliothèque iostream à la place des entrées-sorties dans le style du **C**.

A cause de leur extensibilité les entrées-sorties de <iostream> doivent être préférées à celles de <cstdio> (<stdio.h> en **C**). De plus ces dernières se synchronisent mal avec les entrées-sorties standard.

6.1 Les sorties

6.1.1 Sortie des types prédéfinis

L'opérateur << est utilisé pour envoyer un flux de caractères vers un flot de sortie. Il est prédéfini par surcharge pour tous les types standard. Sa syntaxe est :

flot de sortie << objet

L'opérateur << utilisé est identique à l'opérateur de décalage à gauche, mais se différencie aisément de celui-ci par le type `ostream` de son opérande gauche.

Nous verrons en §6.2.8 comment définir de nouveaux flots de sortie. Deux flots sont prédéfinis : `cout` et `cerr`, qui correspondent respectivement au flot de sortie et au flot d'erreur standard.

L'instruction :

```
int x(1234);
cout << x;
```

envoie sur la sortie standard la suite de caractères `'1','2','3','4'`, qui représentent x en notation décimale. Cet ordre écrit donc `1234`.

`cout <<x` est une expression qui a pour effet de bord d'écrire `x` et a pour *valeur de retour* une référence au flot de sortie `cout`. Nous pouvons utiliser la référence retournée à gauche d'un nouvel opérateur << pour écrire :

```
cout << "x=" << x;
```

à la place de son équivalent :

```
cout << "x=";
cout << x;
```

L'associativité à gauche de << permet d'éviter les ambiguïtés entre sortie et décalage :

```
int x(1234);
cout << 1 << x;   // -> 11234
cout << (1 << x);// -> 2468
```

Outre les types prédéfinis l'opérateur << permet d'écrire des pointeurs sur `void` sous forme d'une adresse mémoire, et donc par conversion il permet d'écrire tout type de pointeur. Les `const char*` sont traités de manière spéciale, l'opérateur << suppose qu'il s'agit de l'adresse d'un tableau de caractères et transfère ses éléments sur le flot jusqu'à ce qu'il rencontre un caractère nul (non compris). Cela permet en particulier d'effectuer des sortie de littéraux chaînes de caractères.

Pour passer à la ligne nous utiliserons le manipulateur `endl` (voir §6.2.7) ; on écrira donc :

```
cout << "x=" << x << endl << "&x=" << &x << endl;
```

6.1.2 Sortie des types utilisateurs

Pour un type utilisateur nous pouvons surcharger l'opérateur de sortie afin d'obtenir un interface homogène.
Exemple :

```
class Rationnel{
public:
  Rationnel(int numerateur, int denominateur);
  int numerateur();
  int denominateur();
  ostream & put(ostream& os) const {
    return s <<  numerateurM << '/' <<denominateurM;
   }
private:
  numerateurM;
  denominateurM;
};
ostream& operator << (ostream& s, Rationnel r){
  return r.put(os);
}
```

Nous utiliserons ces définitions ainsi :

```
Rationnel(2,4);
cout << "r=" << r << endl;  // -> r=1/2
```

Ici l'opérateur `operator<<(ostream& ,Rationnel )` n'utilise pas les membres privés de `Rationnel`, il est donc placé hors de la classe. Cependant comme il fait partie de l'interface de manipulation des rationnels, sa déclaration sera placée dans le même fichier en-tête que la classe `Rationnel`.

Dans le cas ou l'opérateur de sortie accède aux membres privés il doit être déclaré comme fonction amie de la classe. Il est toujours préférable d'éviter l'utilisation de `friend` en utilisant une *méthode* de sortie comme `Rationnel::put`.

6.2 Entrées

6.2.1 Entrées formatées

Les entrées s'effectuent grâce à l'opérateur `>>` que nous utiliserons ainsi :

flot d'entrée >> variable

Cette opération extrait des caractères du flot d'entrée et les convertit dans une valeur du type de la variable placée à droite. L'opérateur `>>` est prédéfini pour les types scalaires standard.

Le flot d'entrée standard est `cin`, nous pouvons donc écrire :

```
int x;
cin >> x;
```

Cette opération lit une suite de chiffres sur l'entrée standard et les convertit en l'entier dont ils sont la représentation (par défaut décimale) ; cet entier est placé dans `x`.

Si le flot d'entrée contient les caractères `" 1234 56 78 "` *(sans les guillemets)*, l'entier 1234 sera placé dans `x`, après quoi le flot d'entrée pointera sur l'espace situé entre 4 et 5. Les espaces, tabulations et fin de lignes sont par défaut sautés par l'opération d'entrée `>>` et servent de séparateurs.

Si nous effectuons sur le même flot les opérations :

```
char a,b;
cin >> a >> b;
```

Nous chargerons le caractère `'5'` dans `a` et `'6'` dans `b`. L'opérateur `<<` est aussi défini pour les types flottants ; il n'est pas défini pour les `void*` car lire *dans une* adresse mémoire n'a généralement pas de sens.[1]

L'opérateur `>>` accepte cependant un argument `char*` , il suppose qu'il s'agit de l'adresse d'un tampon mémoire alloué et suffisamment grand, et il y place les caractères extraits du flot d'entrée jusqu'au séparateur suivant, et ajoute un caractère nul en fin de chaîne.
Exemple :

```
char tampon[10];
cin >> tampon; // dépassement si il y a plus de 9 caractères
```

Il n'y a aucune vérification de la disponibilité d'un tampon suffisamment grand et le dépassement de la zone mémoire allouée peut être catastrophique pour le programme. Dans le cas où on devrait utiliser ce type d'entrée il est prudent d'indiquer la taille du tampon à l'aide du manipulateur `setw`, on écrira donc :

```
#include <iomanip>
const int tailleTampon(10);
char tampon[tailleTampon];
cin >> setw(tailleTampon) >> tampon;
```

[1] `operator >> (void *&)` quant à lui est défini et permet de lire une adresse, il n'est utilisé qu'exceptionnellement (pour un débogueur par exemple).

qui limite à un maximum de 9 le nombre de caractères qui peuvent être extraits par l'opération `>>` ; ce maximum sera remis automatiquement à zéro après l'opération pour indiquer à nouveau qu'il n'y a pas de contrôle.

Mais il est conseillé d'éviter ces entrées vers un `char *` qui sont source de beaucoup d'erreurs. Elles seront remplacées par des entrées non formatées présentées au paragraphe suivant ou, plus souvent, par des entrées de `string` (voir §1.1 et §11).

6.2.2 Entrées non formatées

Les entrées des types prédéfinis utilisent des fonctions de conversion pour transformer les caractères du flot dans une valeur du type. Mais quand nous écrivons des entrées pour les types utilisateurs nous avons souvent besoin de lire le flot d'entrée littéralement avant d'interpréter ces entrées comme une valeur du type que nous définissons.

Les entrées de caractères sans formatage peuvent s'effectuer aussi avec l'opérateur `>>`. Par défaut les espacements servent de séparateurs et sont sautés, mais on peut désactiver ce comportement grâce au manipulateur `noskipws` . Si nécessaire on pourra sauter *à la demande* les espacements avec le manipulateur `ws`. Le manipulateur `skipws` rétablit le comportement par défaut.

```
char tampon[10];
cin >> noskipws >> setw(10) >> tampon;
 ......utilise le tampon
cin >> ws  >> setw(10) >> tampon;
cin >> skipws;
```

La bibliothèque d'entrées-sorties offre quelques primitives spécifiquement destinées aux entrées non formatées.

La méthode `istream& istream::get(char& )` lit le caractère suivant dans le flot d'entrée et le place dans l'argument passé en référence ; `char istream::get( )` effectue la même opération mais a le caractère lu comme valeur de retour. Contrairement à leurs équivalents **C**, ce ne sont pas des fonctions de bas niveau, et employer plusieurs `get` à la place d'une opération plus complexe ne peut provoquer qu'une perte d'efficacité.

Pour lire une suite de caractères dans un tampon nous pourrons utiliser la méthode :

```
istream & istream::get(char * tampon, int n,
                       char delimiteur='\n')
```

Elle lit au plus n-1 caractères dans le flot d'entrée et les place dans le tampon, elle s'arrête *avant* le délimiteur qui reste dans le flot d'entrée. La chaîne est

complétée par un caractère nul. La fonction `getline` a les mêmes arguments et le même comportement toutefois elle lit *aussi* le délimiteur, qui est remplacé par le caractère nul.
Exemple :

```
cin.get(tampon,100,';');
```

lit jusqu'au ' ;' suivant et au plus 99 caractères. Si on applique cet ordre à l'entrée :

```
"  abc;de\nf;  gh\n"
```

il lira `"  abc"` dans le tampon (avec les deux espacements initiaux et le `\0` de fin de chaîne mais *sans* le point-virgule).

Si on effectue tout de suite un second appel, une chaîne vide est lue car le `";"` est en tête de flot. Par contre :

```
cin.get();cin.get(tampon,100,';');
```

lira `"de\nf"` dans le tampon.

Après une opération d'entrée la méthode :

```
int istream::gcount();
```

peut être appelée pour connaître le nombre de caractères qui ont été extraits du flot d'entrée.

Quand nous n'avons pas besoin des caractères entrés, mais nous désirons simplement avancer dans le flot jusqu'à un caractère donné nous utilisons la fonction :

```
istream& ignore(int n=1, int delim = EOF);
```

qui est semblable à la fonction `getline`, mais ignore les caractères entrés. Les caractères sont ignorés jusqu'à ce que soit *n* caractères aient étés lus, soit le caractère `delim` ait été entré. On utilise l'entier maximal `numeric_limits<std::streamsize>::max()` pour avoir la lecture du délimiteur comme seule condition. Remarquons que le délimiteur par défaut est ici la fin de fichier et non la fin de ligne.

6.2.3 Examen anticipé d'un caractère dans un flot d'entrée

Une zone de données peut être délimitée par son dernier caractère, mais très souvent la fin de la zone sera marquée par un caractère qui n'appartient pas à la zone. Par exemple la fin d'un nombre entier est marquée par le premier caractère qui n'est pas un chiffre ; ce caractère n'appartient pas à l'entier. Pour entrer ce type de données nous devons anticiper l'entrée d'un caractère, la bibliothèque `iostream` nous fournit deux possibilités pour cela.

Programme 6.1 lecture d'un `Rationnel`

```
class Rationnel{
public:
  struct  NullDenominateur{};
  ...
  istream& get(istream  &is);
private:
  int numerateurM;
  int denominateurM
};
istream& operator >> (istream  &is, Rationnel& r){
  return r.get(is);
}
Rationnel::get(istream  &is)
  int i,j;
  is >>i >>ws;
  if (is.peek()=='/'){
    is.get();       //lit le "/"
    is >> j;
    if(j==0) throw NullDenominateur();
    int pgcd=arithmetique::pgcd(i,j);
    numerateurM=i/pgcd;
    denominateurM=j/pgcd;
  }else{
    numerateurM=i;
    denominateurM=1;
  }//endif
  return is;
}
```

La fonction membre `peek()` de la classe `istream` rend le caractère suivant en entrée sans avancer dans le flot. Avec `peek()` nous pouvons consulter le caractère sans le consommer. Un `get()` devra être appliqué à sa suite pour effectivement passer au caractère suivant.

Cette méthode devra être employée de préférence à la fonction membre `put-back( char)` qui permet de remettre dans le flot le dernier caractère lu. Il est impossible de remettre plusieurs caractères de suite, ainsi que de remettre un caractère différent de celui qui à été lu.

6.2.4 Entrée d'un type utilisateur

Nous pouvons écrire des entrées pour les types utilisateurs en surchargeant l'opérateur `<<`. Cet opérateurs peut soit accéder aux membres privés ou protégés et être défini comme `friend`, soit utiliser une *méthode* d'entrée.

La solution qui utilise une *méthode* pour accéder aux membres privés est bien préférable car elle assure la localité des données (voir §7.3).

Le programme 6.1 donne un exemple de définition d'une entrée pour un nouveau type.

6.2.5 État d'un flot

Les opérations d'entrées-sorties peuvent échouer pour différentes raisons. Le flot peut être physiquement altéré, parce que le fichier auquel il est lié est détérioré, ou parce que le médium qui le lie au fichier est détérioré par une panne d'un composant matériel ou logiciel.

Plus fréquemment vous essayez de lire des données dans un format qui n'est pas présent dans le flot (comme quand on essaie de lire un entier décimal alors qu'un caractère alphabétique est dans le flot).

Enfin, le flot peut être en fin de fichier, ce qui ne constitue certes pas une *anomalie*, mais doit être reporté au programme utilisateur.

Traitement des erreurs d'entrée/sortie.

méthode	drapeau	signification
`bool ios::eof()`	`ios_base::eofbit`	le fichier est à sa fin après une opération réussie.
`bool ios::fail()`	`ios_base::failbit`	l'opération a échoué.
`bool ios::bad()`	`ios_base::badbit`	le flot est altéré.
`bool ios::good()`	`ios_base::goodbit`	l'opération a réussi sans atteindre la fin de fichier.

TAB. 6.1 – Test de l'état d'un flot

Des méthodes utilitaires pour consulter l'état d'un flot sont définies pour tous les flots dans la classe `ios` et présentée dans la table 6.1.

Ces méthodes testent les *marqueurs d'état* `ios_base::eofbit`, `ios_base::failbit`, `ios_base::badbit` , `ios_base::goodbit` qui sont définis pour tous les flots dans la classe `ios_base` .
Exemple :

```
int i;
cin << i;
if(cin.fail()){
  cout << "caractère:" << cin.get() <<incorrect"<<endl;
}else if(cin.bad()){
  cout << "anomalie sur l'entrée standard"<<endl;
}
```

Nous pouvons nous servir de `ignore` pour sauter les caractères entrés après une erreur d'entrée-sortie.

Par exemple quand le flot d'entrée contient des caractères qui ne correspondent pas au format numérique, un essai de lecture d'un nombre provoque le positionnement du drapeau `fail()`. Nous pouvons indiquer l'erreur en consultant ce drapeau, mais remettre simplement les drapeaux à zéro et essayer une nouvelle lecture ne peut qu'échouer puisque les caractères fautifs sont toujours dans le tampon d'entrée. La solution est soit d'essayer une lecture non formatée ou plus simplement d'ignorer les caractères jusqu'à un point de reprise sûr.
Exemple :

```
float note;
while (true){ //inv: entrée non effectuée
   cout << "Entrez la note" << endl;
   cin >> note;
   if ( cin.good()) { break; } //opération réussie
   cin.clear(); // remet à 0 les indicateurs
   // ignore les caractères jusqu'à la fin de ligne.
   cin.ignore(numeric_limits<std::streamsize>::max() ,'\n');
}
```

Deux opérateurs supplémentaires sont définis dans la classe `ios`

– `operator void*() const;` qui rend `! fail()`.
– `bool operator! () const;` qui retourne `fail()`.

Grâce à ces opérateurs quand on place une référence à un flot à l'intérieur d'un test il est converti dans la valeur booléenne qui correspond à `!fail()`. Il est donc possible d'avoir des expressions condensées telles que :

```
char c;
  if(! cin.get(c)){
    cout "erreur d'entrée" << endl;
  }
```

Exceptions pour des erreurs d' entrée sortie.

Si les erreurs d'entrée d'un utilisateur ne peuvent être considérées comme *exceptionnelles* et doivent donc être traitées par des alternatives du flot de contrôle normal du programme, d'autres erreurs d'entrée ou de sortie sont causées par la défaillance d'un composant matériel ou logiciel et peuvent être traitées par le mécanisme des exceptions.

La méthode `exceptions` de la classe `ios_base` nous permet de demander qu'une exception soit levée quand certaines erreurs d'entrée-sortie se produisent.

Par exemple

```
cin.exceptions(ios_base::badbit|ios_base::failbit);
```

demande que l'exception `ios_base::failure` soit levée si l'opération a échoué sur le flot d'entrée (`failbit`) ou il est dans un état altéré (`badbit`).

Cela nous permettra de contrôler le cas ou une lecture a été tentée avec un format incompatible avec les caractères effectivement présents sur le flot d'entrée.

La déclaration :

```
cout.exceptions(ios_base::badbit|ios_base::failbit|
                ios_base::eofbit);
```

est utile car elle permet de traiter les cas d'erreur en **sortie** qui sont souvent négligés car peu fréquents et dépendants de facteurs externes. Pourtant des médias de sortie peuvent devenir indisponibles, des fichiers corrompus, des liaison NFS s'altérer ; des systèmes de fichier peuvent être à court de place ou d'*inodes*. Un système qui doit être robuste ne peut ignorer ces possibilités d'erreurs.

Un appel à `exception` sans argument retourne les drapeaux d'état qui sont positionnés pour déclencher une exception.

6.2.6 Contrôle du formatage des sorties.

L'utilisateur a parfois besoin de formater ses sorties suivant des normes précises. Il doit contrôler l'alignement des nombres, la taille de la zone de sortie, le format pour l'écriture des nombres, etc. Quelques aspects des entrées peuvent aussi être paramétrés.

Le formatage est contrôlé par une série de drapeaux de la classe `ios_base` donnés dans la table 6.2. Les drapeaux de la classe `ios_base` peuvent être combinés à l'aide des opérateurs bit à bit habituels `| & ~ !` . On se servira surtout du `or` (`|`) pour combiner les drapeaux. Les changements de drapeaux persistent jusqu'à ce qu'on demande à nouveau leur changement.

`dec`	sortie en décimal
`fixed`	sortie en virgule fixe
`hex`	sortie en hexadécimal
`internal`	remplissage entre signe et valeur
`left`	cadrage à gauche
`oct`	sortie en octal
`right`	cadrage à droite
`scientific`	sortie en flottant
`showbase`	préfixer par 0 en oct. et 0x en hex
`showpoint`	écrire les zéros après la virgule
`showpos`	utiliser + pour les entiere positifs
`skipws`	sauter les espaces en entrée
`unitbuf`	vider le flot après chaque opération
`uppercase`	'E', 'X' plutôt que 'e', 'x'
`adjustfield`	drapeau de cadrage
`basefield`	drapeau de base entière
`floatfield`	drapeau des sorties flottantes

TAB. 6.2 – Drapeaux de `ios_base`

`fmtflags flags() const`	lit les drapeaux
`fmtflags flags(fmtflags fmtfl)`	positionne les drapeaux
`fmtflags setf(fmtflags fmtfl)`	ajoute des drapeaux
`fmtflags setf(fmtflags fmtfl,fmtflags mask)`	ajoute des drapeaux
`void unsetf(fmtflags mask)`	efface des drapeaux
`streamsize precision() const`	précision des flottants
`streamsize precision(streamsize prec)`	positionne la précision
`streamsize width() const`	largeur minimale
`streamsize width(streamsize wide)`	positionne la largeur
`static bool sync_with_stdio(bool sync = true)`	synch. avec `stdio`

TAB. 6.3 – méthodes de manipulation des drapeaux

Le tableau 6.3 donne les méthodes de la classe `ios_base` qui permettent de manipuler les drapeaux et de consulter et positionner la largeur de sortie et la position des flottants.

Pour les options qui concernent le cadrage `adjustfield`, le format des entiers `adjustfield`, et le format des flottants `adjustfield` on utilisera la seconde forme du `setf` où la catégorie concernée vient dans le second paramètre `mask`.

On notera que contrairement aux autres méthodes, les changements de `width` ne concernent que l'opération de sortie qui suit.

Ces fonctions sont utiles surtout pour consulter l'état des drapeaux, le sauvegarder, et le restaurer. Pour effectuer des ajustements il est bien plus clair de se servir des manipulateurs de la section suivante §6.2.7.

6.2.7 Les manipulateurs

Les manipulateurs sont des objets que l'on peut placer dans une opération sur un flot, et qui correspondent à une *action* effectuée sur ce flot.

La méthode `operator<<` ou `operator>>` est surchargée pour recevoir ces manipulateurs ; quand elle reçoivent un manipulateur, elles appellent l'action associée sur l'objet flot auquel elles sont reliées. Un manipulateur sans argument est donc simplement un pointeur sur une fonction et l'opérateur `operator <<` ou `operator >>` appelle cette fonction. Les manipulateurs avec arguments sont de conception un peu plus compliquée : ce sont des objets, appelés *objets fonctions*, d'une classe patron. L'usage des fonctions objets est expliqué en 10.4 .

L'emploi des manipulateurs est très simple et ne demande pas de comprendre le mécanisme parfois compliqué de leur fonctionnement. Une liste des manipulateurs qui peuvent être passés à un flot de sortie est donnée figure 6.4 et une liste des manipulateurs d'un flot d'entrée figure 6.5.

Exemple :

```
cout << 12 << ',' ;          // 12,
cout << hex << 12 << ',';    // c,
cout << oct << 12 << ',';    // 14
cout << endl << setw(4) << setfill('#') << '/' <<12 <<'/'<< endl;
                             // /12##/
cout << right <<setfil('*') << internal;
cout << setw(6)<< '/' <<-12 <<'/'<< endl;
                             // /-***12/
```

boolalpha	écrit les booléens de manière symbolique.
noboolalpha	écrit les booléens comme 0 ou 1.
dec	sortie des entiers en décimal.
endl	écrit un '\n' sur un flot de sortie et le vide.
ends	écrit un '\0' sur un flot de sortie et le vide.
fixed	écrit les flottants avec un point décimal.
flush	vide le flot de sortie.
hex	sortie des entiers en hexa.
internal	espace entre le signe et le premier chiffre.
left	cadre le nombre à gauche de la zone d'écriture.
oct	sortie des entiers en octal.
right	cadre le nombre à droite de la zone d'écriture.
scientific	écrit les flottants avec un exposant.
setfill(int c)	remplace le caractère de remplissage par c.
setprecision(int)	nombre de positions décimales des flottants.
setw(int)	taille de la zone de sortie, pour la prochaine opération
showbase	écrit 0 (resp. 0x) avant un nombre octal (resp. hexa).
noshowbase	n'écrit pas 0 ou 0x avant un nombre octal ou hexa.
showpoint	écrit les zéros après le point décimal.
noshowpoint	n'écrit pas les zéros après le point décimal.
showpos	écrit + pour un nombre positif.
noshowpos	n'écrit pas de signe pour les nombres positifs.
skipws	passe les espacements.
noskipws	ne passe pas les espacements.
uppercase	écrit X en hexa. et E et en repr. scientifique.
nouppercase	écrit x en hexa. et e et en repr. scientifique.

TAB. 6.4 – manipulateurs de sortie

boolalpha	lit les booléens comme true ou false.
dec	entrée des entiers en décimal.
hex	entrée des entiers en hexadécimal.
noboolalpha	lit les booléens comme 0 ou 1.
oct	entrée des entiers en octal.
setw(int)	nombre maximal de caractères lus par la prochaine opération.
tie(os)	vide le flot de sortie os à chaque entrée.
ws	saute les espacements.

TAB. 6.5 – manipulateurs d'entrée

Programme 6.2 Liaison d’un fichier avec un flot

```
int main(){
  ofstream sortf("dec.txt");
  if (! sortf.is_open()) {
     erreur ("dec.txt ne peut être ouvert");
  }
  ifstream entf("oct.txt");
  if (!entf.is_open()){
    erreur("oct.txt introuvable");
  }
  entf >> dec;   //entrée en décimal
  sortf << oct;  // sortie en octal
  while(!entf.bad() && ! sortf.bad()) { //jusqu'à fin de fichier
    entf >> i;
    sortf <<i;
  }
  if (sortf.bad()){
    erreur("problème sur sortie sortf.txt");
  } else if (!entf.eof()){
    erreur("erreur sur entrée entf.txt");
  } else {
    cout << "transcription en octal de dec.txt";
    cout << " sur oct.txt effectuée"<<endl;
  }
}
```

6.2.8 Liaison d’un fichier avec un flot

Il est possible d’employer un flot d’entrée `ifstream`, de sortie `ifstream`, ou d’entrée-sortie `fstream` qui accède à un fichier. La bibliothèque `<fstream>` fournit les fonctions nécessaires.

Le nom du fichier doit être donné à la construction ou l’ouverture du fichier.

On peut préciser un mode d’ouverture avec un second argument définit dans la classe `ios_base` et qui sont décrits figure 6.6.

```
ofsream outf( "monFichier.txt’’, ios_base::app);
```

On pourra vérifier que le fichier a effectivement pu être ouvert avec la méthode `is_open()`. La vérification de l’état du flot et du succès des opérations est effectuée en consultant les méthodes `good()`, `fail()`, `eof()`, `bad()` décrites §6.2.5
.

`app`	*append* ajouter aux données présentes
`ate`	*at end* aller en fin de fichier
`binary`	E/S binaires au lieu de texte
`in`	en lecture
`out`	en écriture
`trunc`	tronquer à zéro

TAB. 6.6 – modes d'ouverture des fichiers

La fermeture du fichier est généralement effectuée à la destruction de la variable par le destructeur de *stream* ; Mais il est possible d'appeler `close()`.

La liaison d'un fichier à un flot est illustrée dans le programme 6.2

6.2.9 Liaison d'un flot avec un `string`

Il est possible de lire et d'écrire dans une chaîne `string` en utilisant les flots. L'en-tête `<sstream>` définit des flots de sortie `ostringstream` et des flots d'entrée `istreamstring` liés à des chaînes de caractères.

Un `ostringstream` croît à la demande, il n'y a donc pas à se préoccuper d'une taille de tampon. Ils sont utiles pour formater des messages.

Un `istringstream` lit depuis le `string` qui l'initialise.

L'utilisation de ces deux classes est illustrée par le programme 6.3.

Programme 6.3 Liaison d'un flot avec un `string`

```
string erreur ( string cause, int ligne, int pos)
{
      ostringstream ost;
      ost << "erreur: " << cause;
      ost << "ligne: " << ligne <<"pos: " << pos;
      return ost.str();
}

string traduire( string ent)
{
      istringstream ist(ent);
      ostringstream ost;
      string mot;
      while (ist >> mot){
        if(mot == "bleu" ){
          ost << "rouge"
        }else{
          ost << mot;
        }
      return ost.str();
}
```

Chapitre 7

Classes et Objets

7.1 Le modèle objet

Dans le modèle objet nous effectuons l'abstraction en reconnaissant des objets dans le domaine étudié. Ces objets coopèrent en utilisant les services des autres objets et en fournissant leurs propres services au monde extérieur. Quand deux objets coopèrent le fournisseur de service est nommé *serveur* et l'utilisateur du service est nommé *client*. Un objet offre des services grâce à des *opérations* ou *méthodes* qui peuvent fournir une information quant à l'état de l'objet ou modifier cet état. Chaque *méthode* a une *signature* comprenant son nom, le type de ses arguments et le type de la valeur retournée. Elle comporte une spécification abstraite de l'opération qu'elle effectue, comprenant une *pré-condition* qui donne l'ensemble des états de l'objet dans lesquels on peut appliquer l'opération et une *post-condition* qui précise l'état de l'objet après l'opération.

L'ensemble des opérations offertes par un objet, identifiées par leurs signatures et accompagnées des conditions de leur appel, constitue le **protocole** de l'objet.

7.1.1 Les langages objets.

Il est possible d'implémenter une conception par objets dans un langage classique. Par exemple le système de fenêtrage *X Windows* est réalisé en **C**. Il est cependant souhaitable de disposer d'un **langage objet** qui offre un support syntaxique pour :

- regrouper les objets de même protocole ,
- cacher aux utilisateurs les détails de la représentation de l'objet,
- vérifier qu'une opération n'est appliquée que si elle figure dans le protocole de l'objet,
- permettre à des objets de partager la partie commune de leur protocole.

Date
-anM: Annee -moisM: Mois -n0jourM: int
+create() +create(a:Annee,n0j:int,:) +create(a:Annee) +create(a:Annee,m:Mois,j:int=1) +jour(): Jour +n0jour(): int +n0jourAn(): int +mois(): Mois +annee(): Annee +incr(nbjours:int=1): void +put(os:ostream &): ostream &

FIG. 7.1 – Diagramme UML de la classe Date

– s'assurer qu'une opération n'est appelée que sur un objet dans un état approprié.
– vérifier que les opérations sont conformes à leur spécification

Ces possibilités ne sont pas toutes offertes par les langages objets disponibles.

Nous allons examiner les possibilités que nous offre **C++** pour implémenter une conception objet.

7.1.2 Classes

Les classes dans le *modèle objets* permettent de regrouper des objets de même protocole, et sont le support de l'*abstraction*. Elles permettent à l'utilisateur de manipuler des objets d'une même classe d'une manière uniforme en fonction de leur protocole. La notion de classe nous permet aussi de réaliser l'*encapsulation* en cachant les détails de l'objet aux utilisateurs.

7.1.3 Les méthodes

Les *méthodes* ou *fonctions membres* correspondent à un service demandé par un client de l'objet. Cette demande de service comprend l'identité de l'objet concerné, l'identité de la méthode invoquée, et la valeur des arguments transmis par le client à l'objet. L'objet répond à cette demande en transmettant une réponse sous forme d'une valeur de retour et facultativement en changeant son état.

Les paramètres et la valeur de retour de la fonction membre doivent être d'un type correspondant à la signature de la méthode.

Certaines méthodes ne changent jamais l'état de l'objet, elles sont déclarées comme constantes en les faisant suivre du mot clé `const`. (voir §7.3.3)

Généralement l'argument passé à la méthode sert à paramètrer la demande de service. alors la fonction membre ne doit pas changer l'état de l' objet passé en paramètre.

Nous conviendrons donc de ne passer des arguments à une méthode que par valeurs, références constantes ou par pointeurs constants. (voir règle 7.8 p. 74)

Certaines fonctions membre, et en particulier certains constructeurs, peuvent accepter en paramètre une référence à une ressource acquise par l'objet ou partagée avec d'autres objets. Elle sera dans ce cas passée par référence ou pointeur non constant. (voir à ce sujet la section §7.4.6).

7.1.4 État d'un objet.

L'effet des opérations que fournit un objet dépend de son *état*.

Une voiture ne peut accélérer que si son moteur est en marche, et l'effet d'une action sur l'accélérateur dépend de l'*état* de la voiture. Cet état pour décrire le comportement de la voiture nous devrons le prendre en compte sous la forme de vitesses de rotation, de pente, de direction, de vitesse de déplacement, de vitese de vent, ... Si je dois écrire une classe qui représente des voitures je dois trouver des moyens de *représenter* cette état, les rotations moteurs, les vitesses de vent, les passagers deviennent alors des *long double* ou encore d'autres classes. Mais le comportement de la voiture ne dépend pas de la taille d'un *double* sur ma machine pas plus que de la fréquence du processeur.

L'état utilisé lors de l'analyse et qui me permet de spécifier mes opérations est une représentation abstraite, mathématique, indépendante des variables utilisées concrètement pour représenter l'objet.

Si, par exemple, nous définissons une classe «tableau d'entiers» nous pourront définir un *état abstrait* d'un objet comme étant une fonction partielle à domaine et image entiers. Cet état abstrait peut nous servir à spécifier toutes les opérations sur un tableau. En revanche il serait erroné d'utiliser pour cela des adresses mémoires car nous ne souhaitons pas que le comportement du tableau varie avec son emplacement mémoire.

7.1.5 Encapsulation et qualité

Les utilisateurs des services d'un objet peuvent accéder au protocole de cet objet. Pour que la classe puisse fournir un comportement de l'objet conforme aux spécifications de ce protocole nous devons coder l'état de l'objet au moyen de données. Mais ces données ne sont qu'un artifact nécessaire à la représentation de l'objet et l'utilisateur ne doit jamais pouvoir y accéder directement.

Ces besoins sont la source du classement des membres de la classe en deux catégories l'*interface* et l'*implémentation*.

L'encapsulation nous permet de restreindre l'accès aux objets d'une classe à l'interface de cette classe. Les interfaces seront conçues et spécifiées de manière à ne dépendre que de l'état abstrait des objets.

Ce qui veut dire concrètement que les variables d'état ne doivent pas figurer dans la spécification publique d'une opération. Sinon il serait impossible à l'utilisateur de traiter les objets de manière abstraite.

L' implémentation fournira une représentation concrète de l'état au moyen des objets membres de la classe. Dans l'exemple du tableau elle donnera un moyen de représenter la fonction partielle grâce à une structure mémoire convenablement allouée en mémoire.

Utilité de l'abstraction Nous manipulerons donc les systèmes complexes par des représentations abstraites, indépendantes des détails internes. Ces abstractions sont moins volatiles que les mécanismes internes qui contribuent à leur réalisation. Grâce à elles nous pouvons atteindre un seuil de fiabilité et de qualité qui est requis à la fois pour des critères économiques et fonctionnels.

Dans un logiciel important les implémentations des différentes classes doivent pouvoir être modifiées à moindre frais, pour tenir compte de l'évolution du système dans le temps et de sa diffusion sur de nouvelles plates-formes et à de nouveaux utilisateurs. Il est donc essentiel que l'interface ne dépende pas de la représentation concrète de l'état par l'implémentation.

Dans l'exemple `Date` (figure 7.1), le changement de la représentation interne de la date pour un numéro annuel de jour ne devrait pas demander aux clients de la classe `Date` de modifier leurs programmes. Pour cela nous plaçons les attributs dans la partie privée, et nous spécifions les méthodes d'une manière abstraite, indépendante de la représentation choisie pour les dates. `Date`.

7.1.6 Spécification de la classe et de ses méthodes

pré-conditions et post-conditions Chaque classe est définie lors de la phase de conception avant d'être programmée dans le langage cible choisi.

Le document de conception doit mentionner pour chaque méthode une spécification de la pré-condition et la post-condition de cette méthode (cf §7.1).

Cette spécification abstraite devrait accompagner chaque méthode : elle permettra la validation et guidera les tests. Cependant il sera encore plus utile de représenter dans le langage objet la pré-condition sous la forme d'une expression booléenne. Nous pourrons en incluant cette expression dans le langage de programmation vérifier explicitement que la méthode est appelée dans les conditions

prévues.

La programmation d'une post-condition est souvent plus difficile car elle peut demander d'effectuer à nouveau, *et par un moyen sûr* le travail même de la méthode ; dans certains cas cependant la post-condition ou une condition un peu plus faible peut être testée. Et nous avons alors un moyen pendant le débogage de vérifier que la méthode fait effectivement le travail prévu.

Invariant Il n'y a généralement qu'une partie des états des variables de la classe (état *concret*) qui constitue un état valide. Le prédicat qui lie ces états est appelé *invariant de la classe*. L'invariant doit être conservé par toutes les méthodes de la classe (sauf le destructeur).

Dans la classe `Date` (programme 7.1) les jours, mois, années sont représentés par des entiers qui doivent vérifier un *invariant* pour représenter une date valide.

On inclura des ordres conditionnels qui, pendant le débogage :
- vérifient la pré-condition à l'appel des méthodes.
- vérifient la post-condition au retour des méthodes.
- vérifient l'invariant au retour des méthodes sauf pour le destructeur.

Rec. 10.6 Spécifiez les classes en utilisant des pré-conditions, post-conditions, exceptions et invariants de classe.

Rec. 10.7 Utilisez **C++** pour décrire les pré-conditions, post-conditions, et invariants de classe.

7.2 Codage des classes en C++

Une classe est un type de donnée **C++** . Comme tout objet **C++** une classe doit faire l'objet d'une *déclaration* et d'une *définition*.

La déclaration de la classe donne son nom, et la définition de la classe donne le nom et type des objets et fonctions membres qui la composent (c'est à dire la *déclaration* de ces objets et fonctions).

La déclaration et la définition[1] de la classe sont usuellement effectuées conjointement[2] et placées dans un fichier en-tête (`.hh`).

La définition des méthodes de la classe est placée dans un fichier implémentation (`.cc`).

---

[1]Bien qu'il s'agisse à la fois d'une déclaration et d'une définition, il est fréquent de parler de *déclaration de classe*

[2]Il est aussi possible d'effectuer une *déclaration anticipée* d'une classe, qui est une déclaration sans définition

Programme 7.1 En-tête C++ de la classe `Date`

```
typedef int Annee;
enum Jour {lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche};
enum Mois {janvier, fevrier, mars, avril, mai, juin, juillet,
           aout, septembre, octobre, novembre, decembre};
class Date {
public:
      struct MauvaiseDate{}; //exception
      // date par défaut 1/1/2000
      Date() throw ();
      // si 0<=n0j<=365 (ou 366 an biss.) crée la date
      // sinon lance une exception MauvaiseDate
      Date(Annee a, int n0j) throw(MauvaiseDate);
      // 1/1/a
      explicit Date(Annee a) throw(MauvaiseDate);
      // si j/m/a existe crée la date correspondante
      // sinon lance une exception MauvaiseDate
      Date(Annee a, Mois m, int j=1) throw(MauvaiseDate);
      // **** ACCESSEURS *****
      // retourne le jour de la semaine
      Jour jour() const throw ();
      int n0jour()const throw (); //n0 du jour dans le mois
      int n0jourAn()const throw (); //n0 du jour dans l'année
      Mois mois() const throw ();
      Annee annee() const;
      // **** MODIFICATEURS ****
      // incrémente la date de nbjours (1 par défaut)
      void incr(int nbjours=1) throw(MauvaiseDate) ;
      // ***** E/S *****
      std::ostream & put (std::ostream & os)  const throw();
private:
    Annee anM;
    Mois moisM;
    int n0jourM;
};
```

La définition de la classe spécifie principalement l'interface de la classe[3], la définition des méthodes l'implémentation. Le fichier «en-tête» d'une classe sera aussi appelé «fichier interface».

La définition de la classe est effectuée par un bloc précédé du mot-clé `class`.

```
class nom classe { déclaration des membres ... };
```

La figure 7.1 donne un exemple de définition d'une classe.

Il est utile de se souvenir qu'il faut terminer le bloc par un `;`, c'est une source d'erreurs de compilation qui déconcerte parfois les débutants en **C**++ .

7.2.1 Parties publiques et privées d'une classe

Chaque classe comprend un *interface* qui est introduit par le mot clé `public` et qui donne les déclarations des données et fonctions membres publiques. Le terme *fonction membre* dans le vocabulaire **C**++ est synonyme de méthode.

Une fonction membre `public` est une méthode qui fait partie du protocole de l'objet et peut donc être appelée par tout possesseur de l'identité de l'objet.

La partie publique peut en outre en **C**++ comprendre des objets membres, possibilité discutée en §7.2.4.

Le mot clé `private` introduit la partie *privée* qui consiste dans la déclaration des objets membres qui permettent de représenter l'état de l'objet, on parle aussi parfois d'*implémentation* de l'objet, mais ce mot trop utilisé est imprécis.

On peut aussi placer dans la partie privée des *fonctions privées* qui ne peuvent être appelées que par l'objet lui même ou un objet de la même classe.[4]

Certaines classes ou fonctions peuvent aussi être déclarées *amies* (`friend`) (cf. §8.2.4) et avoir ainsi accès aux membres privés.

Dans une classe les membres sont par défaut privés, mais il est recommandé d'indiquer explicitement les spécificateurs d'accès.

Style A.12 Donnez toujours un spécificateur d'accès pour les classes de base et les données membres.

Puisque le mot-clé `public` introduit l'interface utilisateur de la classe, on commencera toujours par cette section.

Style A.13 Les sections `public`, `protected` et `private` doivent être placées dans cet ordre.

Si le mot `class` est remplacé par le mot `struct` nous définissons une *structure* qui, en **C**++ n'est autre qu'une classe où les membres sont par défaut publics (voir §7.7).

---

[3] au moins pour la partie publique, voir le paragraphe suivant

[4] En **C**++ contrairement à d'autre langages la protection se fait par classe et non par objet.

7.2.2 Conventions de codage des classes

Il est utile, pour documenter ses programmes, de choisir une convention typographique qui permette de reconnaître quelle sorte d'entité représente un nom. Dans la suite nous utiliserons comme noms de classe des noms dont l'initiale est majuscule et les lettres suivantes minuscules, ou mélangeant minuscules et majuscules conformément aux conventions de style A.3, A.4 et A.6 page 21 .

Pour distinguer les données membres on utilisera aussi une convention :

Style A.5 Les données membres sont postfixées par la lettre M.

Une autre convention au paragraphe §7.3 donnera un nom aux fonctions liées à la manipulation de ces donnée membres.

Des conventions strictes de codage ont été rendues indispensables dans les gros progiciels qui devaient suppléer l'absence d'un langage fortement typé pour supporter leur codage.

Les boites à outil de fenêtrage «Athena» ou «Motif» qui bien qu'ayant une conception objet sont écrites en C, donnent de bons exemples de codage.

Plus récemment des projets importants comme *kde* ou *gnome* imposent une discipline stricte de codage, et fournissent la documentation et les outils pour la mettre en oeuvre.

Ces conventions restent très utiles même avec les langages fortement typés, car quand bien même le compilateur connaît précisément le type d'une expression, le programmeur lui-même ne dispose que d'une petite mémoire parfois volatile, et ne sait gérer qu'un contexte réduit.

Cette discipline de codage est le seul moyen de développer du logiciel en équipe. Elles est encore plus importante quand l'équipe est sujette à des changements de personnel, ou quand la maintenance est assurée par une équipe distincte de celle qui a effectué le développement. Dans tous les grands services fournissant des logiciels, des **normes** régissent tout le processus de développement.

On consultera aussi à ce propos la section §2.1.1, ainsi que les recommandations 1.1 à 1.4 et toutes les conventions de style dans l'annexe §A.

7.2.3 Accès à un membre d'un objet

Les références aux données membres se font par leur simple nom à l'intérieur des méthodes de la classe. Ces noms désignent alors les parties correspondantes de l'objet courant. De même l'appel d'une fonction membre depuis une autre fonction membre de la même classe applique cette fonction à l'objet courant.
Exemple :

```
Date::Date(....){
 ....
 nojourM=...  // nojourM de la Date en construction
}
Date::mois()const{
  return moisM; //moisM de l'objet courant;
}
```

En dehors de la classe il faut préfixer le nom de membre par l'identité (nom, pointeur ou référence) de l'objet concerné.
Exemple :

```
Date vac(12, février, 1994);
vac.incr();
```

Toute fonction membre a donc, outre ses arguments déclarés, un argument supplémentaire implicite constitué par une référence à l'objet sur lequel elle s'applique. La méthode `f1` suivante :

```
class C {
...
D f1( A, B );
};
```

est donc équivalente, en ignorant les règles de portées, à la fonction :

```
D f2(C&, A, B);
```

L'appel `c.f1(a,b)` correspond à `f2(c,a,b)`.

Des méthodes de même prototype peuvent se trouver dans plusieurs classes.

Dans la notation `objet.membre(...)` `membre` est recherché parmi la classe de `objet`.

Pour *définir* une fonction membre, il faut utiliser le nom de la classe pour préfixer le nom de la fonction, on écrit donc :

```
Mois Date::mois() const {return moisM;}
```

De même pour appeler une fonction statique ou désigner un objet statique nous avons besoin de nommer explicitement la classe en utilisant l'opérateur `::`, ils seront donc notés :

```
  Classe::Membre
```

7.2.4 Visibilité des attributs

En conception un attribut public est un objet membre que les utilisateurs de la classe peuvent *consulter*. Mais **C**++ ne fait aucun contrôle sur les objets membre déclarés `public`, les utilisateurs peuvent donc non seulement les consulter mais aussi les modifier.

Nous devront donc adopter des normes de codage pour donner à ces objets le comportement désiré.

*Ce qui est conçu comme un attribut public n'est pas codé en **C**++ comme un objet* `public`.

Règle 10.1 Déclarez `private` les données membres.

Dans l'exemple `Date` précédent nous avons placé les données membres `n0jourM`, `moisM`, `anM` dans la partie *privée* de la classe. Si nous les avions déclarées publiques tout utilisateur d'une `Date` aurait pu changer et modifier les données de `Date` sans passer par les méthodes.

7.3 Accesseurs et modificateurs

Puisque les attributs sont privés il nous faudra souvent des méthodes qui permettent de les consulter et de les modifier. Les premières seront nommées *accesseurs* et les secondes *modificateurs*.

Il est utile d'avoir une convention fixe qui lie les données à leur accesseur et leur modificateur. La convention proposée ici est de nommer la donnée `choseM`. La fonction membre *accesseur* qui en donne la valeur s'appelera `chose( )`, et la méthode *modificateur* qui permet de la modifier aura aussi pour nom : `chose(Tchose nouvChose )`.

Une autre convention courante est de nommer la donnée `chose`, l'accesseur `get_chose( )` et le modificateur `set_chose( )`.(voir aussi §2.1.1 page 20).

Utiliser des accesseurs et des modificateurs pour les attributs, nous permet de satisfaire trois impératifs de la conception objet qui sont :

- Préserver la cohérence des données.
- Faciliter les vérifications et les tests.
- Localiser les dépendances.

Avant d'examiner la mise en œuvre des accesseurs et modificateur il convient de mettre en garde le lecteur contre la tentation d'accompagner *chaque* attribut d'un accesseur et d'un modificateur. Les membres ne sont la que pour coder l'état abstrait de l'objet et c'est *cet état abstrait* auquel on accède et qu'il convient de modifier. Et la cohérence des attributs, c'est à dire l'*invariant* de l'objet ne peut

souvent pas être décomposé en une conjonction de prédicats dont chacun ne porterait que sur une variable.

Donc dans de nombreux cas les modificateurs de l'objet ne seront pas des modificateurs d'une variable isolée mais plutôt des modificateurs d'un groupe indivisible de variable.

Préserver la cohérence des données. Si le concepteur souhaite permettre de modifier les jours, mois, ans nous devons nous soucier de la cohérence des données qui constituent la date. Les types énumération semblent nous protéger contre les mois et jours qui ne figurent pas au calendrier. Mais si nous permettons l'accès au jour et mois il nous faudra accepter que le mois de février puisse compter 31 jours, ce que notre calendrier n'admet pas.

Si nous voulons autoriser la modification des `Date`s, nous fournirons donc deux méthodes de modifications où *modificateurs* dans la partie publique :

```
void jour(Jour nouvJour) throw (MauvaiseDate); //change le jour
void mois(Mois nouvMois) throw (MauvaiseDate); //change le mois
bool an(Annee nouvAn) throw (MauvaiseDate);    //change l'année
```

Nous pouvons ainsi vérifier que les dates restent cohérentes.

Mais bien que tout entier soit acceptable comme numéro d'année , seuls certains mois, certaines années peuvent comporter des jours de numéros 29 à 31. La méthode `an(int nouvAn)` doit s'assurer de la cohérence de la date obtenue.

Cependant une telle programmation montre un défaut de conception, car un utilisateur doit pour effectuer un changement global vers une date existante tenir compte des dates intermédiaires qu'il traversera. Ainsi nous devons pour aller du 31 mars au 1 Avril changer d'abord le mois puis le jour, le contraire provoquant une exception.

Plutôt qu'une modification individuelle des champs, une conception correcte offrira une méthode :

```
// Change la date pour nouvJour nouvMois nouvAn et la retourne
// si la nouvelle date n'est pas valide lance une exception
// MauvaiseDate
 Date changeDate(Jour nouvJour, Mois nouvMois, Annee nouvAn)
                                              throw(MauvaiseDate );
```

Ou encore :

```
// Change la date pour  nouvDate et la retourne
 Date changeDate (const Date& nouvDate) throw(MauvaiseDate );
```

Et ici nous pouvons nous reposer sur le contrôle effectué par le constructeur qui a déjà vérifié la nouvelle date, nous changeons une date valide pour une nouvelle date valide.

Le chapitre 7.6.2 nous proposera d'écrire cette méthode :

```
Date& operator = (const nouvDate&)throw(MauvaiseDate );
```

Faciliter les vérifications et les tests. Garder privées les données membres nous aide aussi à vérifier, tester et déboguer le programme car les erreurs sont plus facilement localisables.

Nous pouvons ajouter aux modificateurs une vérification des pré-conditions de la méthode qui permet de s'assurer qu'elle est appelée dans les cas prévus, et vérification de l'*invariant*), qui assure qu'elle ne laisse pas l'objet dans un état incohérent. Ainsi la méthode `incr` pourra s'écrire :

```
void incr( int nbjours=1){
  assert(nbjours>=0); // pré-condition
   ....               // corps de incr
  assert(OK());       // invariant
```

Un libre accès aux données laisse le programmeur désarmé quand des bogues se produisent dans un grand programme, l'erreur peut venir de partout. La seule possibilité est de la rechercher avec un débogueur symbolique ce qui comporte une part de tâtonnement qui croit avec l'importance des procédures. Et quand bien même une erreur est trouvée d'autres erreurs similaires peuvent rester longtemps cachées dans le code.

Notre discipline de programmation nous permet de rechercher les erreurs dans les quelques modificateurs par inspection du code, par la technique des invariants mentionnée ci-dessus, et en dernier recours en plaçant avec un débogueur symbolique des *breaks* à l'entrée et à la sortie de chaque méthode de modification.

L'utilisation des accesseurs pour donner l'état de l'objet est discutée plus en détail paragraphe §7.5.4.

Localiser les dépendances. Ce dernier argument concerne surtout les membres qui sont des objets de classe

Rendre publique une donnée membre revient à ajouter au protocole de l'objet tout le protocole de ce membre, et donc à déclarer que tout utilisateur de la classe est un utilisateur de cet objet membre. Cela implique une liaison forte entre la classe utilisatrice et la classe du membre, puisque toute modification du protocole du membre se traduit par la modification du protocole de la classe qui le contient.

Les objets membres publics propagent donc les mises à jour vers les clients de la classe qui contient le membre.

Si un objet membre est public son protocole appartient à la description abstraite de la classe, or souvent les objets membres sont des codages d'un état abstrait et devraient être cantonnés dans l'implémentation. Les objets publics ne sont généralement présents que par une négligence du codage.

7.3.1 L'exception des classes concrètes

Certaines classes ou structures qui sont elles-mêmes contenues dans la partie privée d'une autre classe, ou dans l'implémentation d'un paquetage. Elles constituent des **classes concrètes** qui, n'ont qu'une classe (ou un paquetage) client et n'ont pas d'interface abstrait.

Ces structures constituent simplement des composants pré-assemblés avec lesquels nous bâtissons nos abstractions. Il est naturel de laisser le libre accès a leurs membres puisqu'une classe ou un paquetage prend en charge leur protection.
Exemple :

```
class Date{
struct DonnéesDate{
    DonnéesDate(int, Mois, int);
    int n0jourM;
    Mois moisM;
    int anM;
};
public:
  Date( DonnéesDate dataN);
  .....
private:
  DonnéesDate dataM;
};
```

L'héritage nous permettra aussi, plutôt que de placer une `DonnéesDate` comme donnée membre, de faire hériter `Date` de `DonnéesDate` en écrivant :

```
struct DonnéesDate{ ... };
class Date: private DonnéesDate { ...... }
```

Dans les deux cas l'accès aux membres de `DonnéesDate` est limité par le `private` de `Date`.

7.3.2 Référence à l'objet lui même

À l'intérieur d'une classe nous aurons rarement besoin de manipuler globalement l'objet courant auquel s'appliquent les fonctions membres. Nous n'avons besoin habituellement que de ses membres. Parfois cependant nous aurons besoin d'un pointeur ou une référence à l'objet courant, en particulier quand la syntaxe de **C**++ nous impose de rendre une référence à un objet que nous avons modifié (voir §7.6.2 et §7.6.8).

Le mot clé `this` désigne un pointeur constant sur l'objet sur lequel la méthode a été appelée. Il pourra aussi être déréférencé par l'opérateur `*` pour obtenir une référence à l'objet courant.

```
Date& Date::operator += (int i){
    for (int j(0); j<i; j++) incr(); // this->incr();
    return *this;  //référence à l'objet courant
}
```

7.3.3 Objet constant et fonction membre constante

Le mot clé `const` à la suite d'une fonction membre indique qu'elle ne change pas l'état de l'objet. En conséquence pourra s'appliquer à des objets déclarés constants par le même mot clé `const`. Les objets déclarés constants ont donc un protocole qui est différent des objets non constant.

Le concepteur doit penser en définissant une classe que cette définition est en vérité double. Elle définit à la fois la classe des objets non constants et la classe des objets constants. Le protocole de ces deux classes doit être défini.

A moins d'être surchargée par une méthode constante ; une méthode non constante s'applique aussi aux objets constants. Le protocole des objets constants est donc un sous ensemble de celui des objets non constants.
Exemple :

```
const Date fête_nationale(1789, juillet, 14);
......
cout << fête_nationale.n0jour()  <<endl;
......
fête_nationale.incr();// erreur
```

Comportement standard des objets constants. Le comportement le plus simple pour un objet constant est d'avoir ses propres membres constants. Le compilateur

vérifiera donc que tous les membres sont bien constants dans le corps d'une fonction constante, c'est à dire qu'il n'est appliqué que des méthodes constantes aux objets de classe et que les objets de type prédéfini ne peuvent pas être modifiés par une affectation.

Pour le compilateur, alors qu'une méthode `f(B b)` de la classe `A` est analogue à une fonction :

```
f(A& objet, B b)}
```

la méthode `f(B b) const` est analogue à

```
f(const A& objet, B b)}
```

Certaines fonctions doivent dans tous les cas être déclarées `const` ; ce sont celles qui ne changent pas l'état du programme, ni l'état de l'objet sur lequel elles sont appelées, ni l'état d'un autre objet.

Règle 7.11 Une fonction membre qui ne change pas l'état du programme doit être déclarée `const`.

Protocole des objets constants. Nous devrons adapter le comportement du compilateur à nos besoins de conception. Un objet constant est un objet qui ne change pas d'état *abstrait*.

Dans les cas simples il est possible de déclarer comme constantes les méthodes qui ne changent pas les objets membres. C'est, sauf exception, un minimum que l'on peut demander à une fonction `const`. C'est aussi ce qui est vérifié par le compilateur.

Cela ne suffit pas quand certains membres sont des pointeurs ou des références. ou quand les méthodes changent une ressource fournie par un paramètre.

Dans tous les cas on suivra la règle :

Règle 7.12 Une fonction qui donne un accès non `const` à la représentation d'un objet ne doit pas être déclarée const.

Quand une méthode déclarée constante a un argument non constant cela lui permet de changer l'objet passé en paramètre, et si cette donnée fait partie des membres de l'objet courant alors nous modifions l'objet courant que la méthode soit ou non déclarée constante. On doit donc éviter de déclarer constantes de telles méthodes.

Le passage d'un objet en paramètre non constant correspond à l'acquisition, l'utilisation ou la libération d'une ressource et ces opérations ne peuvent être déclarées constantes.

C'est au niveau de la spécification et de la conception que devront être définis les sémantiques des objets constants. Le compilateur vérifie par défaut que l'on ne change pas les membres, mais ce comportement ne convient pas dans tous les cas.

Quand l'objet contient un pointeur, nous souhaitons généralement qu'un objet constant ne change pas *l'objet pointé* alors que le compilateur vérifie seulement qu'il ne change pas *le pointeur*.

La sémantique du protocole `const` est délicate en cas de partage d'objets membres, nous devons définir soigneusement en quoi consiste la vue de chacun sur les objets partagés.

Dans les cas les plus courants on peut se limiter *à ne rien changer* et appliquer ainsi de manière littérale la règle :

Rec. 7.13 Ne laissez pas les fonctions membres `const` changer l'état du programme.

Nous pouvons être ainsi certain que le comportement du programme ne changera pas quelque soit le nombre d'appel d'une fonction membre `const`.

Mais dans certain cas cette recommandation ne peut être suivie de manière stricte sur l'état *concret* même si elle reste vraie pour l'état *abstrait*.

C'est en particulier le cas quand une donnée est calculée lors de son accès puis mémorisée comme le montre l'exemple 7.2. Cette classe à une stratégie de déclenchement d'exceptions opposée à celle de `Date`, alors que `Date` lance une exception si on essaye de *construire* une date erronée `DateBrute` laisse librement construire toute `Date`, mais lance une exception si on essaie d'*utiliser* une date erronée.

Ici `correcte` et `vérifiée` peuvent être changés même pour un objet `DateBrute` constant. Quand la sémantique d'une classe implique que l'on doit modifier un membre d'un objet `const`, on aura recours à une donnée `mutable` si le compilateur implémente cette notion, sinon à une conversion explicite de type qui devra être documentée.

Règle 6.4 Déclarez les données membres `mutable` si elles doivent être modifiées par une fonction membre constante.

7.3.4 Fonction en ligne

Certaines méthodes correspondent à des petites fonctions pour lesquelles le coût de l'appel serait disproportionné avec l'importance du corps de la fonction.

Une fonction est déclarée *en-ligne* en la précédant du mot `inline`. Son appel est réalisé par l'expansion en ligne du corps de la fonction. Cependant contrairement aux macro-instructions le compilateur vérifie le typage pour les fonctions en ligne comme pour les fonctions compilées.

Programme 7.2 classe `DateBrute`

```
class DateBrute {
public:
    struct MauvaiseDate{}; //exception
       //crée la date sans vérification
    DateBrute (Année a, Mois m, int j=1) throw();
       // le jour de la semaine exception si non valide
    Jour jour() const throw(MauvaiseDate);
       //n0 du jour  exception si non valide
    int n0jour()const  throw(MauvaiseDate);
       // mois pour une date valide exception sinon
    Mois mois() const  throw(MauvaiseDate);
       // année si valide exception sinon
    Année année() const throw(MauvaiseDate);
       //change le mois
    void mois(Mois) throw();
       //change le jour;
    void jour(Jour) throw();
       //change l'année;
    void Année( int) throw();
       //la date est elle valide?
    bool valide();
private:
    int n0jourM;
    Mois moisM;
    int anM;
    mutable bool correcte;
    // mutable car positionnée lors de l'accès
    mutable bool vérifiée;
    // vérifie la date et retourne sa validité,
    // positionne correcte et vérifiée
    bool vérifie();
};
void DateBrute::mois(Mois m){
  moisM = mois;
  vérifiée = false;
}

bool DateBrute::valide(){
  return (vérifiée &&  correcte) || vérifie();
}

Mois DateBrute::mois(M)const{
  if ( ! valide() ) throw MauvaiseDate();
  return moisM;
}
```

Rec. 7.1 Déclarez `inline` les fonctions simples.

Lors de l'appel d'une fonction il faut effectuer un changement de contexte ce qui implique la création en mémoire d'une structure d'exécution pour la nouvelle fonction. Pour cela il faut généralement faire un appel au gestionnaire mémoire.

Quand les fonctions sont simples elles sont donc plus efficaces quand elles sont déclarées en-ligne, quand elles sont *très* simples la taille du code peut même diminuer car le code généré est inférieur à celui de la préparation de la structure d'exécution. Au contraire pour des fonctions compliquées la taille du code généré par une fonction en-ligne devient inacceptable, allonge exagérément le temps de la compilation et finit par dégrader aussi les performances quand le code devient trop grand pour tenir dans une page mémoire.

En programmation-objet les accesseurs à des données membres sont de bon candidats pour la directive `inline`, de même que les modificateurs quand il n'y à pas de vérification compliquée des arguments d'entrée.

Une méthode définie à l'intérieur de la définition de classe est considérée comme *en-ligne* par défaut.

Ce mode de définition est bien adapté à l'illustration d'un exemple court dans un ouvrage de programmation, en revanche, elle offre le double désavantage, dans un logiciel, de montrer à l'utilisateur des informations privées et de rendre malaisé le passage de *en-ligne* à *compilé*.

Style A.15 Définissez les fonctions `inline` en dehors de la définition de classe.

Les fonctions en-lignes seront placées dans un fichier `.icc` (voir la convention de style A.10 p.81 et le paragraphe §4.1.2). Les fonctions `inline` n'ont qu'une portée de fichier et doivent être définies dans l'unité de compilation où elles sont déclarées, ce que nous effectuons en incluant le fichier `.icc` à la fin du fichier en-tête.

Dans sa phase d'optimisation le compilateur développe en ligne les courtes fonctions, le programmeur est donc partiellement libéré de la gestion des `inline`, cependant cela ne s'applique qu'aux fonctions que *voit* le compilateur, il ne peut pas développer en ligne les fonctions des autres unités de traductions auquel il n'a pas accès. L'utilisation d'un fichier des `inline` (`.icc`) inclus après l'en-tête permet au compilateur de développer en ligne les méthodes venant des autres en-têtes. Cela à cependant le défaut d'introduire dans tous les fichiers compilés une dépendance avec le `.icc` : tout module incluant le `.hh` dépend aussi du `.icc`.

7.3.5 Surcharge des méthodes

Les fonctions membres, comme toutes les fonctions, peuvent être surchargées. (voir §3.4.1) La surcharge des fonctions est utilisée pour indiquer que ces fonc-

tions effectuent des tâches similaires avec des arguments différents. En revanche des fonctions dont le rôle est différent doivent aussi avoir des noms différents.

Règle 7.14 Toutes les variantes des fonctions membres surchargées doivent être utilisées pour le même usage et avoir un comportement similaire.

Pour éviter de multiplier les méthodes surchargées on aura recours aux arguments par défaut (voir §3.4.2). Rappelons la règle 7.17 (p. 76) qui demande de placer toujours ces défauts avec la déclaration de fonction, c'est à dire dans la déclaration de classe.

7.3.6 Membre statique

Une donnée ou une fonction membre peut être déclarée `static`, elle ne dépendra alors pas d'un objet particulier. Il n'existe qu'une seule copie d'une donnée statique dans une classe, c'est pourquoi les objets et méthodes statiques sont aussi appelés *objets de classe* ou *méthodes de classe*.

Une méthode statique ne peut accéder qu'aux membres statiques de la classe. On accède aux fonctions et données statiques en les préfixant par le nom de la classe suivi de l'opérateur `::` .

Les constantes statiques ne doivent pas être définies et initialisées à l'intérieur de la déclaration de classe mais séparément dans l'implémentation. Le mot clé `static` ne doit pas être répété pour la définition.
Exemple :

```
//   Date.hh
class Date{
....
static int nbjourMois[decembre+1];
....
};

// Date.cc
int Date::nbjourMois[decembre+1]=
                     {31,28,31,30,31,30,31,31,30,31,30,31}
```

7.4 Constructeurs et Destructeurs

Tout objet de classe est *construit* lors de sa définition. La construction d'un objet permet de donner à l'objet un *état initial*. Cette initialisation s'effectue par

un *constructeur*. De la même manière les objets seront détruits par l'intermédiaire d'un *destructeur*. Constructeurs et destructeurs sont des méthodes spéciales de la classe, qui peuvent être définies explicitement par le programmeur ou, en l'absence de définition, être générées par le compilateur.

7.4.1 Constructeur

Toute initialisation d'un objet de classe se fait par un constructeur qui est une fonction membre et est généralement fournie par le programmeur. Chaque classe peut fournir un ou plusieurs constructeurs qui permettent de créer l'objet en initialisant ses objets membres. Ce sont des fonctions membres qui portent le même nom que la classe à laquelle elles appartiennent et qui n'ont pas de type de retour (même pas `void`).

Toute construction d'objet commence donc par l'initialisation des objets membres. Une valeur initiale pour un objet membre sera spécifiée en faisant suivre son nom d'une liste d'initialiseurs entre parenthèses.

Les membres objets de classes dont l'initialisation n'est pas mentionnée sont initialisés par le constructeur sans argument *quand celui-ci existe* ; on dit alors qu'ils sont initialisés *par défaut* ou qu'il s'agit d'une *initialisation implicite* . Quand il n'existe pas d'initialisation par défaut, ne pas initialiser l'objet membre est une erreur détectée par le compilateur . Les membres types prédéfinis qui ne sont pas initialisés explicitement, reçoivent une valeur par défaut (*pour un objet global*), ou restent non initialisés (*pour un objet* automatique) ; ce cas est l'objet de la section §7.4.2 et du programme 7.4.

Les initialisations sont suivies de l'exécution du corps du constructeur.

Les constructeurs des objets membres sont exécutés dans l'*ordre des déclarations de ces objets* (indépendamment de l'ordre d'appel des constructeurs).

Dans le programme 7.3, `débutM`, `finM`, `libbelléM` sont initialisés *dans cet ordre* dans les quatre constructeurs de `Période`.

Le constructeur de la ligne 22 est donc correct : bien que `finM` apparaisse avant `débutM`, il sera initialisé ensuite, et l'expression `débutM+durée` est correctement évaluée.

Le constructeur de la ligne 25, est incorrect car quand `débutM` est initialisé, `finM` est encore inconnu et l'expression `finM-durée` ne peut être évaluée. `finM` devrait être initialisé avec `fin-durée`.

Le constructeur de la ligne 28, peut ou non provoquer des erreurs suivant l'interface de `Date`. Comme `débutM` et `finM` ne sont pas explicitement initialisés, le compilateur va essayer de les initialiser avec le constructeur `Date::Date()` si celui-ci existe. Si ce constructeur par défaut n'existe pas, il reportera une erreur.

Dans tous les cas il n'est pas possible en lisant ce code de décider si le manque d'initialisation est intentionnel ou le résultat d'un oubli.

Programme 7.3 classe Période

```
class Période{
public:
    // période commençant à début, finissant à fin
    Période (Date début, Date fin, string libellé);
    // période de durée jours, commençant à début.
    Période (Date début, int durée, string libellé);
    // période de durée jours, finissant à fin
    Période ( int durée, Date fin, string libellé);
    // période d'un seul jour.
    explicit Période (Date jour);
    ........
    autres méthodes
    ...............
private:
    Date débutM;
    Date finM;
    std::string libbelléM;
};
Période::Période (Date début, Date fin, string libellé):
    débutM(début),finM(fin),libelléM(libellé){
}
Période:: Période (Date début, int durée, string libellé):
    finM(débutM+durée),débutM(début),libelléM(libellé){
}
Période::Période ( int durée, Date fin, string libellé):
    finM(fin),débutM(finM-durée),libelléM(libellé){//Erreur
}
Période::Période (Date jour, string libellé):libelléM(libellé){
   débutM=jour; finM=jour;  //Erreur?
}
```

Si nous voulions vraiment utiliser le constructeur sans argument il faudrait écrire :

```
Période::Période (Date jour,string libellé):
    débutM(),finM(),libelléM(libellé){ ... }
```

Mais bien sûr ici, il est plus direct d'écrire :

```
Période::Période (Date jour,string libellé):
    débutM(jour),finM(jour),libelléM(libellé){ ... }
```

Pour éviter les erreurs dues à un ordre d'initialisation non souhaité on se conformera à la règle :

Règle 5.6 Ordonnez la liste des initialiseurs dans le même ordre que celui de l'en-tête ; d'abord les classes de bases, puis les données membres.

Pour fournir des constructeurs acceptant différents types ou nombres d'arguments, nous définirons plusieurs constructeurs qui formeront une famille de fonctions surchargées.

Pour diminuer le nombre des constructeurs on peut utiliser des arguments par défaut.[5]

7.4.2 Construction explicite ou implicite

Dans l'exemple de l'initialisation de `Période` précédent on ne voit pas clairement si l'initialisation par défaut de `débutM` et `finM` est souhaité ou si c'est le fruit d'un oubli.

Plus gênant encore est le cas du programme 7.4 qui initialise parfois la classe `EntierA` à 0 parfois à une valeur arbitraire. On évitera ambiguïtés et erreurs en suivant la recommandation :

Rec. 5.5 Initialisez toutes les données membres.

Puisque tout objet membre doit être construit il est préférable de donner toujours une valeur initiale, et non de laisser le compilateur essayer une valeur par défaut, pour ensuite changer la valeur dans le corps du constructeur ou, pire, laisser une variable d'un type prédéfini dans un état arbitraire.

Comme le montre le cas de la classe `EEntier` dans le programme 7.4 les erreurs se propagent à partir des classes qui oublient les initialisations vers les classes qui les emploient comme membres, ou qui en dérivent.

[5] Il faut cependant prendre garde à ne pas introduire une conversion implicite dangereuse (voir §7.6.6).

Programme 7.4 Initialisation implicite de membres

```
#include <iostream>
#include <algorithm>
#include <iterator>
using std::cout; using std::endl;

class EntierA{
   public:
      EntierA(){} //pas d'init. explicite de valM
      int val(){return valM;}
   private:
      int valM;
};
class EntierB{
   public:
      EntierB():valM(){} //init. explicite par défaut de valM
      int val(){return valM;}
   private:
      int valM;
};

class EntierC{
   public:
      EntierC():valM(0){} //init explicite à 0 de valM
      int val(){return valM;}
   private:
      int valM;
};

class EEntier{
   public:
      EEntier():valM(){}
      int val(){return valM.val();}
   private:
      EntierA valM;
};
template <typename InputIterator>
void ecrit(InputIterator deb, InputIterator fin); //  voir prog. 10.17
```

Programme 7.5 Initialisation implicite (prog. principal et résultats)

```
EntierA eA0;
EEntier ee0;

int main(){
   EntierA eA1;
   EntierB eB1;
   EntierC eC1;
   EEntier ee1;
   EntierA eaT[10];
   EntierA* eaU=new EntierA[10];
   cout << "EntierA eA0 global: "<<eA0.val()<<endl;
   cout << "EntierA eA1 local: "<<eA1.val()<<endl;
   cout << "EntierB eB1 local: "<<eB1.val()<<endl;
   cout << "EntierC eC1 local: "<<eC1.val()<<endl;
   cout << "tableau eaT local: ";
   ecrit(eaT,eaT+10);
   cout << "tableau eaU sur le tas: ";
   ecrit(eaU,eaU+10);
   cout << "EEntier ee0 global: "<<ee0.val()<<endl;
   cout << "EEntier ee1 local: "<<ee1.val()<<endl;
}
```

Résultat

```
EntierA eA0 global: 0
EntierA eA1 local: 134522000
EntierB eB1 local: 0
EntierC eC1 local: 0
tableau eaT local: 1074156096, 1108542220, -1073745672,
134522888, 134543360, 1108542220, 1073819680, 1074694022,
-1073745624, 1073787440,
tableau eaU sur le tas: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
EEntier ee0 global: 0
EEntier ee1 local: 1
```

C'est donc une bonne discipline de programmation de toujours mentionner la valeur initiale de tous les objets membres dans chaque constructeur. Quand bien même ces initialiseurs sont identiques à ceux qui sont générés automatiquement, les mentionner permet d'indiquer que cette initialisation est voulue. Cela évite aussi d'avoir à rechercher une hypothétique initialisation lors de la relecture du code ou son débogage.

Dans certains cas la valeur initiale d'un membre demande à être calculée avant l'initialisation et il est souvent possible de le faire dans une fonction statique utilitaire. Dans le cas d'une initialisation complexe dans laquelle interviennent les différents membres il est aussi possible de leur donner d'abord une valeur transitoire avant d'effectuer les calculs dans le corps de la méthode de construction[6].

7.4.3 Constructeur généré par le compilateur

Un *constructeur par défaut* est un constructeur sans arguments. Le compilateur génère quand cela lui est possible un constructeur par défaut `A()` pour une classe `A` **qui ne définit aucun constructeur**. Ce constructeur construira chaque objet membre avec son propre constructeur par défaut (ou son initialisation par défaut pour un type prédéfini).

En revanche pour une classe qui définit un constructeur le compilateur ne génère jamais de constructeur par défaut. Certaines classes n'ont pas de constructeur par défaut et tous leurs objets doivent être initialisés explicitement.

Il est préférable de toujours définir un constructeur ; seules les structures concrètes *à la* C qui ont leurs membres publics auront recours aux constructeurs par défaut.

7.4.4 Destructeur

Un destructeur pour une classe `A` a pour nom `~A()`. Il permet à un objet de signaler sa disparition à son environnement. Le principal rôle du destructeur est de libérer les ressources acquises par l'objet et en particulier la place qu'il occupait. Le destructeur ne prend jamais d'argument.

La destruction d'un objet comporte trois étapes : appel du destructeur, destruction des membres, destruction des classes de base.

Quand un objet est détruit, si un destructeur est défini, il est d'abord appelé, puis dans tous les cas ses objets membres sont détruits dans l'ordre inverse de leurs déclarations.

Quand ces objets membres sont des objets de classe leur destruction suit le même schéma et la destruction se propage de la classe vers les objets contenus.

[6] Mais on évitera toute acquisition de ressource dans le corps du constructeur voir §7.4.6

Programme 7.6 Exemple de destructions

```
#include <iostream>
using std::cout; using std::endl;

class A {
  public:
    A(int i):valM(i){
      cout<< "A("<<i<<")"<<endl;
    }
    virtual ~A(){
      cout<< "~A("<<valM;
      cout<<")"<<endl;
    }
  private:
    int valM;
};

class B:public A {
  public:
    B(int i, int j);
    static int nombre();
    ~B(){
      cout <<"~B"<<endl;
      --nb_objets;
    }
  private:
    A valM;
    static int nb_objets;
};
```

```
int B::nb_objets=0;
int B::nombre(){
  return nb_objets;
}
B::B(int i, int j):A(i),valM(j)
{
  cout<< "B("<<i<<", ";
  cout<< j <<")"<<endl;
  ++nb_objets;
}
int main(){
  {
     B b1(1,7);
  }
  cout << B::nombre()<<endl;
  B* pb=new B(1,7);
  cout << B::nombre()<<endl;
  delete pb;
  cout << B::nombre()<<endl;
  return 0;
}
```

Résultat

```
A(1)
A(7)
B(1, 7)
~B
~A(7)
~A(1)
0
```

```
A(1)
A(7)
B(1, 7)
1
~B
~A(7)
~A(1)
0
```

Quand la classe hérite d'autres classes, alors après la destruction des membres la destruction de l'objet se continue par la destruction des classes de base dans l'ordre inverse de leurs déclarations . Là encore il y a propagation des destructions des classes dérivées vers les classes de base.

Le programme 7.6 illustre l'ordre des destructions.

Les destructeurs sont indispensables pour des objets créant des structures en mémoire libre comme expliqué ci-dessous §7.4.5. Un destructeur virtuel est aussi nécessaire pour toute classe ayant une méthode virtuelle voir §8.4.6. Enfin les destructeurs seront utilisés pour libérer les ressources acquises dans un constructeur comme expliqué en §7.4.6.

7.4.5 Construction et destruction d'un objet

Objet déclaré comme variable locale

Le constructeur d'une variable locale est exécuté chaque fois que le flux de contrôle passe par la définition de la variable, et le destructeur est appelé chaque fois que le flux de contrôle sort du bloc de la définition de la variable. Les destructions sont effectuées dans l'ordre inverse des constructions.
Exemple :

```
void f(){
  Date d1(1787, juillet, 14);
  Date d2(1788, juillet, 14);
  Date d3(1789, juillet, 14);
  for (int i(0); i<50; i++){
    Date d(d3);
    ..........
    }
  ...........
}
```

à chaque appel de la fonction `f()` les dates `d1` puis `d2` et `d3` sont construites. À chaque passage dans la boucle `for` la date `d` est construite (par le constructeur de copie `Date::Date(const Date &)`), il est détruit en sortie de boucle. À la sortie de la fonction les destructeurs de `d3`, de `d2` puis de `d1` sont appelés.

Tableau d'objets d'une classe.

Si aucun initialiseur n'est précisé chaque objet d'un tableau d'objets de classe est initialisé avec le constructeur par défaut. Il est aussi possible de préciser une

liste d'initialiseurs donnant les arguments pour la construction de chaque objet appartenant au tableau.
Exemple :

```
Date tabDate1[20];
Véhicule tabDate2[]=
  {Date(1803,aout,15),Date(1803,septembre, 7)};
```

Les 20 dates de `tabDate1` sont initialisés avec `Date()`.

Dans le cas particulier des tableaux l'initialisation doit se faire avec le signe `=`.

Objet membre

Chaque fois qu'un objet de la classe est construit tous ses membres sont construits. Si le constructeur de la classe prévoit un initialiseur spécifique il est utilisé, sinon le compilateur vérifiera la disponibilité d'un initialiseur par défaut qui sera alors utilisé.

Les membres de classe sont détruits lors de la destruction de l'objet qui les contient et *après* la fin de l'exécution du destructeur quand celui-ci existe.

Variable statique

Les constructeurs des variables statiques globales sont exécutés dans l'ordre des définitions, et avant l'exécution de la fonction `main()`. Les destructeurs des variables statiques sont exécutés après la sortie de `main()` dans l'ordre inverse des constructeurs.

Les variables statiques globales permettent d'effectuer des initialisations et destructions de données avant et après l'appel de `main()`. Exemple :

```
ostream& s= cout<< "Le programme va bientôt commencer" <<endl;
main(){
........
}
```

Les variables statiques des classes sont allouées et initialisées dans les modules objets qui contiennent leurs définitions, lors de l'initialisation de la classe. Pour éviter de dupliquer ces objets, leurs définitions ne doivent pas être effectuées dans les fichiers en-têtes.

Construction et destruction des objets en mémoire libre

Les constructeurs et destructeurs des objets en mémoire libre doivent être appelés explicitement par `new` et `delete`. Comme tout constructeur il devront comporter les arguments nécessaires à l'initialisation.

C'est une erreur d'appeler plusieurs fois `delete` sur un même pointeur, ou sur un pointeur ne correspondant pas à un objet alloué par `new` (ou par `new []` pour `delete[]`).

Aucune destruction n'est effectuée automatiquement par le compilateur, il peut donc subsister des objets en mémoire qui ne sont plus accessibles car les pointeurs sur ces objets ont été détruits.

Contrairement aux langages comme lisp, scheme, caml, java, smalltalk qui récupérent automatiquement un espace mémoire sur lequel il n'existe plus aucun pointeur aucune stratégie *ramasse-miettes* n'est prédéfinie en **C**++ [7].

Il est donc important de détruire les objets en mémoire libre qui ne sont plus référencés.

Quand on détruit la dernière référence à un objet en modifiant la valeur d'un pointeur ou en le détruisant nous devons libérer la mémoire qu'occupe l'objet.

En particulier une classe qui a un membre pointeur et crée des objets en mémoire libre devrait posséder une méthode de destruction.

Quand un objet en mémoire libre est partagé, c'est-à-dire quand on peut y accéder par plusieurs pointeurs, on désignera si possible un propriétaire de l'objet qui est responsable de sa création et destruction. Quand ce n'est pas possible une politique d'accès partagé sera mise en oeuvre. La méthode la plus commune est celle du compteur de références qui permet à chaque client qui libère l'objet de déterminer si il doit être détruit.

L'emploi direct des objets en mémoire libre doit être évité, car il est source de *fuites mémoire*. Pour les conteneurs on utilisera les classes de la *STL*, quand on doit stocker l'adresse d'un objet dans une classe on aura recours à une classe de *pointeurs intelligents* qui assurera la gestion de la mémoire, on consultera à ce propos les sections §7.4.6 et §7.6.5

7.4.6 Acquisition de ressource

Certains objets doivent lors de leur construction acquérir des ressources. Ces ressources peuvent être de la mémoire, des fichiers, des accès à des bases de données, des accès réseaux ect. Ces ressources doivent être libérées par le destructeur. Un exemple est donné dans le le programme 7.7

Ce programme essaie par trois fois d'obtenir de la mémoire dans le corps de son constructeur et la libère dans son destructeur.

[7] Une application peut réaliser un ramasse-miettes en surchargeant l'opérateur `new`.

Programme 7.7 Acquisition peu fiable de ressources

```
#include <cstring>
inline char* c_strcpy(const char * p){
   return std::strcpy(new char [std::strlen(p)+1],p);
}

class Personne1 {
   public:
      Personne1(const char* nom,
               const char* prenom, const char* adresse)
      {
         nomM=c_strcpy(nom);
         prenomM=c_strcpy(prenom);
         adresseM=c_strcpy(adresse);
      }
      ~Personne1()
      {
         delete[] nomM;
         delete[] prenomM;
         delete[] adresseM;
      }
   private:
      char* nomM;
      char* prenomM;
      char* adresseM;
};

```

Tant que la mémoire est disponible, cette stratégie convient. Supposons maintenant que nous obtenions la mémoire pour le nom et le prénom mais qu'il ne reste plus de mémoire disponible pour l'adresse : le constructeur est alors interrompu par une exception `std::bad_alloc` (voir §7.4.5) ; comme l'objet n'est pas complètement construit le destructeur n'est pas appelé, et la mémoire demandée pour le nom et le prénom ne pourra pas être récupérée.

Placer les acquisitions de ressource dans le corps du constructeur est donc une stratégie non fiable.

Nous pouvons essayer de modifier le programme comme en 7.8

Mais ce programme est encore défectueux, quand une exception interrompt la construction de `adresseM` le destructeur de `Personne2` n'est toujours pas appelé, et quand à `nomM`, et `prenomM`, ce sont des pointeurs et n'ont donc pas de destructeur.

Programme 7.8 Autre acquisition peu fiable de ressources

```
class Personne2 {
   public:
      Personne2(const char* nom,
                const char* prenom, const char* adresse):
         nomM(c_strcpy(nom)),
         prenomM(c_strcpy(prenom)),
         adresseM(c_strcpy(adresse))
      {
      }
....
}
```

Là est la solution de notre problème chaque membre qui correspond à l'acquisition d'une ressource doit appartenir à une classe qui a un destructeur qui libère la ressource. Cette technique est connue par la formule «*acquisition de ressources par initialisation*» .

Si nous remplaçons les `char*` par une classe `Chaîne` nous obtenons le programme : 7.9

Dans ce programme, si une exception intervient lors de la construction de `adresseM`, ni le destructeur de `Population3`, ni celui de `adresseM` n'est exécuté puisqu'ils ne sont pas construits, mais ceux de `nomM` et `prénomM` le sont et ils libèrent la mémoire précédemment acquise.

Pour que cela soit possible la classe `Chaîne` doit supporter la technique d'acquisition de ressource par initialisation. C'est le cas des classes `string` de la STL mais aussi la classe du programme 7.10 qui enveloppe la gestion d'un `char*` dans une classe.

7.5 État d'un objet

7.5.1 Deux vues de l'état d'un objet.

Les objets que nous reconnaissons dans le domaine de l'application ont un *état*. L'état est formé par toutes les propriétés de l'objet et leurs valeurs courantes. L'état d'un objet composé d'autres objets est généralement la concaténation (ou plus formellement le produit cartésien) des états de ses composants. Par exemple l'état d'un `Véhicule` est composé de l'état de son `Modèle`, de son `Assurance`, de son `Immat`riculation, de sa `Personne` propriétaire.

Alors que les classes complexes sont construites avec des objets des classes de

Programme 7.9 acquisition par initialisation

```
class Personne {
   public:
      Personne(const char* nom,
                 const char* prenom, const char* adresse):
         nomM(nom),
         prenomMprenom),
         adresseM(adresse)
      {
      }
      ~Personne()
      {
// delete est fait par Chaine
      }
   private:
      Chaine nomM;
      Chaine prenomM;
      Chaine adresseM;
};
```

Programme 7.10 Une classe Chaîne

```
class Chaine{
public:
   Chaine() throw(std::bad_alloc):pM(c_strcpy("")){}
   Chaine( const Chaine& s) throw(std::bad_alloc):pM(c_strcpy(s.pM)){}
   explicit Chaine( const char* s) throw(std::bad_alloc):pM(c_strcpy(s)){}
   const char * c_str() const throw() {return pM;}
   unsigned longueur() const const throw(){return std::strlen(pM);}
   bool operator == (const Chaine& s) const const throw(){
      return (std::strcmp(pM, s.pM)==0);}
   Chaine& operator = (const Chaine& s) throw(std::bad_alloc);
   std::ostream& put(std::ostream& os)const throw(){
      return os << pM;}
   ~Chaine() throw() {delete[] pM;}
private:
   char * pM;
};
std::ostream& operator << (std::ostream& os, const Chaine& s);
};
```

base, celles-ci sont réalisées avec des variables de types prédéfinis. L'état de ces variables forme un *état concret* distinct de l'*état abstrait* vu par le concepteur.

Quand nous avons dans l'application un «modèle» de produit, ce «modèle» est distinct du pointeur sur un caractère, ou de l'entier qui va servir à le coder, et nous ne pourrons probablement pas dire que deux objets sont de même modèle si leur représentation contient le même pointeur. Il sera encore plus absurde de chercher si un modèle est postérieur à l'autre en comparant les valeurs de ces pointeurs.

Pour des classes qui représentent des types simples et sont réalisées avec des structures de données [8] la conception de la classe doit fournir une représentation abstraite de l'état d'un objet. Cette représentation abstraite est une description, généralement formelle, qui doit correspondre à ce qu'on peut observer de l'objet dans le domaine de l'application.

Par exemple on décrira de manière *abstraite* une chaîne de caractères comme une suite d'éléments d'un ensemble des caractères.[9] Cette description est indépendante d'une représentation particulière des chaînes, car l'utilisateur de chaînes de caractères ne souhaite ni traiter de l'encodage des caractères ni de leur stockage en mémoire.

Nous spécifions les méthodes par une *pré-condition* et une *post-condition*. La pré-condition précise quels sont les états de l'objet dans lesquels la méthode peut être appelée ; alors que la post-condition décrit le changement d'état de l'objet que provoque la méthode et la relation entre leurs valeurs de sortie et l'état initial de l'objet. La représentation *abstraite* de l'état est donc utilisée à la fois pour la pré-condition et la post-condition.

7.5.2 Fonction d'abstraction

Quand la spécification, qui appartient aux stades de l'analyse et de la conception est effectuée, le travail de la programmation peut commencer : il consiste essentiellement à trouver un moyen de coder de manière représentable sur une machine l'état et les opérations définis lors de l'analyse. Ce codage doit toujours être lisible, autrement dit il doit s'accompagner d'un «dictionnaire» et de «règles de grammaire» qui permettent de s'assurer que le langage de programmation est bien une traduction fiable de ce que voulait l'utilisateur et qui est formalisé par le concepteur.

Une fois l'analyse et la conception effectuées le travail de la programmation comporte deux points :

– Fournir une structure de donnée dont la valeur est nommée *état concret*,

[8] De tels types sont généralement nommés *types concrets*

[9] Ou plus formellement comme une fonction dont le domaine est un intervalle entier de type $[1..n]$ et qui prend ses valeurs dans l'ensemble des caractères.

accompagnée d'une fonction dite *fonction d'abstraction* qui associe à un état concret, un état abstrait.
– Donner pour chaque opération un algorithme qui modifie l'état concret conformément à sa spécification abstraite.

Pour la classe `Chaîne` du programme 7.10 la fonction d'abstraction associe à un objet de type `Chaîne` la suite des caractères `p[i-1]` situés avant le premier `'\0'`.[10] Une classe pour laquelle nous pouvons définir cette fonction d'abstraction sera bien plus sûre que les simples pointeurs de caractères utilisés en `C`, en effet pour ceux-ci l'état abstrait n'est défini que quand le pointeur pointe sur un tableau de caractères contenant un octet nul.

Pour la méthode `longueur` nous devrons expliquer comment la manipulation des pointeurs effectuée dans le corps de la fonction donne effectivement le nombre de caractères de la chaîne.

7.5.3 État concret d'un objet

L'*état concret* des objets que nous manipulons dans nos programmes est représenté en mémoire par une structure qui code les différentes valeurs de leurs composants. Le modèle de cette structure est généré par le compilateur ; ses valeurs ne correspondent pas toujours de manière unique à l'état abstrait de l'objet représenté.

Par exemple, dans une même configuration mémoire, des pointeurs différents peuvent correspondre à des chaînes de caractères identiques.

Pour les types prédéfinis l'état concret est simplement la valeur de la variable dans l'ensemble des valeurs possibles, c'est à dire un entier, un caractère ... Pour les objets de classe, l'état de l'objet est formé de la suite (plus précisément du produit cartésien) des états de ses membres.

L'état d'un pointeur est l'identité de l'objet référencé, alors que l'état d'une référence est celui de l'objet référencé. Pointeurs et références se comportent donc différemment en ce qui concerne les états.

7.5.4 Accesseur ou fonction d'état

Nous avons vu à la section §7.2.4 que tous les membres doivent être privés. Il est souhaitable que l'état de l'objet puisse être consulté par l'utilisateur, ne serait-ce que pour le diagnostic des anomalies, même si cela ne fait pas partie des fonctionnalités initialement demandées.

---

[10] Une description mathématique serait : la fonction qui associe à un entier i le caractère $p[i-1]$ pour tout i compris entre 1 et N, où N est le plus petit entier tel que $p[N] ==' \backslash 0'$.

Bien entendu cet état que nous fournissons à l'utilisateur est l'*état abstrait*. Nous devons l'extraire des données membres. Ces informations sont délivrées par *fonctions d'états* qui sont des méthodes *constantes*. Les plus communes d'entre elles rendent l'état codé par un des membres et on les appelle alors *accesseurs*.

Accesseur à une donnée. Pour les données membres qui sont des objets de classe ou des types prédéfinis, les accesseurs sont très simples : ils rendent une *copie* du membre.
Exemple :

```
int Date::nojour() const{
  return noJourM;
}
```

Accesseur à des objets externes. Le cas des pointeurs et des références est plus délicat, mais il n'y a généralement pas lieu de rendre un pointeur ou une référence non constante à un objet externe dans une fonction membre.

La classe `Chaîne` pourrait définir un accesseur à son membre `p` par :
Exemple :

```
char* Chaîne::c_str(){
  return p;
}
```

Mais ici le tableau des caractères bien qu'officiellement privé, devient de fait public. En effet on peut effectuer :

```
Chaine s("toto");
char* p=s.c_str();
p[1]='i';             //tito !
delete[] p;           // Qu'est devenu s!
```

Nous devons penser que tout retour d'un pointeur ou d'une référence crée un partage d'objets qui doit être traité conformément aux techniques de la section suivante §7.6.5.

Ajouter un `const` pour protéger le pointeur comme dans :

```
const char * Chaîne::c_str()const{ return p;}
```

peut rendre plus sûre notre programmation bien que le `const` puisse être contourné par un transtypage et ne protège pas contre les essais de destruction.

L'inconvénient de stocker un pointeur sur un objet membre est que l'utilisateur ne peut connaître sa durée de vie et que l'objet risque d'être détruit de manière inopinée pendant ou avant son utilisation.
Exemple :

```
cont char* p=Chaine("toto").c_str();
.......
cout << p << endl; // toto a peut être disparu ou a changé!
```

Cet exemple nous rappelle une recommandation importante dès que nous manipulons des références ou pointeurs :

Rec. 15.17 Ne vous fiez pas à la durée de vie des objets temporaires.

Un dernier essai serait de *recopier* la chaîne comme dans :

```
char * Chaîne::c_str(){
  return std::strcpy(new char[std::strlen(s)+1],s);
}
```

Il convient alors d'informer l'utilisateur de la fonction que la récupération de l'espace mémoire référencé par le pointeur lui revient.

Rec. 10.2 Si une fonction membre retourne un pointeur ou une référence, vous devez documenter la manière de l'utiliser et sa durée de validité.

Cependant dans un projet important il est illusoire de penser que ces directives seront observées, si on n'utilise pas un moyen d'obliger à leur prise en compte. Ici nous *devons* rendre un objet qui prend en charge la destruction de l'espace qui n'est plus utilisé ; cela peut être un de ces *pointeurs intelligents* évoqués dans le paragraphe suivant ou plus simplement un `string` dont la classe incorpore aussi une gestion mémoire.

Dans tous les projets importants l'utilisation de simple pointeurs et d'allocation mémoire non contrôlée est une cause majeure de fuite mémoires (*memory leaks*) qui sont la cause principale des mauvaises performances et de la faible fiabilité des gros programmes.

Mais peut-être notre principale erreur est-elle dans la conception : avons-nous *besoin* de rendre un tableau de caractères ? Si nous voulons simplement que l'utilisateur puisse consulter les caractères de la chaîne, il est plus approprié de fournir la fonction :

```
char Chaîne::operator[](unsigned int i) const {
  return (i< std::strlen(pM)) ? pM[i]:'\0';
}
```

On peut généralement éviter de donner un accès global à un objet référencé, si on redéfinit l'interface d'accès à cet objet avec des méthodes de la classe qui le possède.

7.6 Opérateurs

Les méthodes qui réalisent la copie, l'affectation et le test d'égalité des objets sont nécessaires pour presque toutes les classes ; il est donc souhaitable de leur donner non seulement une syntaxe identique mais encore une sémantique uniforme. Nous allons les examiner maintenant.

7.6.1 Test d'égalité des objets

L'égalité est codée par surcharge de l'opérateur `==` et sert à comparer l'état *abstrait* de deux objets de la même classe. Nous devons bien entendu définir cet opérateur de manière à ce qu'il soit commutatif, transitif, idempotent, toutes caractéristiques qu'un utilisateur peut raisonnablement attendre d'une égalité.

Puisque les méthodes doivent être définies au niveau abstrait, il est indispensable qu'une même méthode appliquée à deux objets égaux donne le même résultat et laisse les objets dans des états égaux. Par exemple si nous appliquons la méthode `ajouter( "XXXX" )` à deux chaînes identiques nous devons obtenir deux chaînes identiques, même si les deux chaînes initiales sont codées à des places mémoire distinctes.
Exemple :

```
bool Chaîne::operator == (const Chaîne& s)const{
  return (std::strcmp(p, s.p)==0);
}
```

La définition de l'opérateur `==`, nous permet de considérer l'état abstrait d'un objet comme l'ensemble des objets qui sont équivalents selon l'opérateur `==`.

Par exemple deux chaînes de caractères ont même état abstrait si `strcmp` les reconnaît comme égales. Deux dates ont le même état abstrait si la fonction `==` de `Date` retourne `true`.

7.6.2 Affectation d'objets

Pour deux objet `a1` et `a2` l'instruction d'affectation `a1 = a2` doit donner à `a1` le même état *abstrait* que celui de `a2`. Il est possible d'effectuer cette affectation en recopiant l'état concret de chaque membre de l'objet : c'est la solution qu'appliquera par défaut le compilateur.

Quand l'utilisateur ne fournit pas une autre fonction, un opérateur d'affectation est généré en effectuant des affectations membre à membre des composants de l'objet. Quand les composants sont des objets de classe qui ont eux-mêmes une affectation correctement définie, cette stratégie convient.

Pour des composants pointeurs, puisque l'identité seule de l'objet pointé est recopiée, nous obtenons un partage des composants. Cette solution permet d'économiser de la place mémoire et assure la cohérence des informations partagées. Si les informations partagées sont constantes, elle sera facilement adoptée ; dans le cas contraire elle demande un contrôle strict des modifications de l'état des objets et une prise en charge spécifique de sa destruction (voir §7.6.5).

Quand il n'est pas convenablement traité, le partage d'un membre non constant peut conduire à des erreurs d'exécution difficiles à déboguer. Dans les nombreux cas où nous souhaitons que l'affectation conduise à une copie de l'état du membre mais pas à son partage, nous devrons copier l'objet identifié par le pointeur et non dupliquer le pointeur.

Nous appelons *copie superficielle* une méthode de copie qui recopie les pointeurs sans explorer les membres pointés, et *copie en profondeur* la méthode qui recopie chaque objet pointé et crée un nouveau pointeur sur cette copie.

Pour notre classe `Chaîne`, puisque nous n'avons pas pris les moyens permettant de partager des chaînes, nous devons les recopier, soit :
Exemple :

```
Chaîne& Chaîne::operator = (const Chaîne& s){
  if (pM!=s.pM){ //protection pour x=x
    delete[] pM;
    pM=std::strcpy(new char[std::strlen(s.pM)+1],s.pM);
  }
  return *this;
}
```

Le retour d'une référence dans une affectation permet d'utiliser son résultat à gauche d'une autre affectation, ou plus généralement, comme un objet auquel on peut appliquer une nouvelle méthode comme dans :

```
(s1=s2).ajouter("XXX")
```

La stratégie d'affectation membre à membre propage les opérateurs d'affectation dans les composants. Si nous prenons la précaution de définir l'affectation et la copie pour une classe élémentaire concrète comme la classe chaîne, nous n'aurons plus à nous en préoccuper dans une classe qui emploie des composants de type `Chaîne` comme la classe `Personne` du programme7.9 pour lequel on définira le constructeur de copie par :

```
class Personne {
public:
  Personne(const Personne& p):
     //copie par recopie des membres
    nomM(p.nomM),prénomM(p.prénomM), adresseM(p.adresseM){
  }
```

La copie est ici équivalente à la copie par défaut et aurait pu être omise ; de même l'affectation par défaut convient.

Nous aurions en revanche dû prendre en charge les copies et affectations si nous avions représenté les différents membres par un pointeur sur une zone en mémoire libre ou comme un tableau de caractères. Rappelons aussi que nous avons vu au paragraphe 7.4.6 que cette méthode est déficiente en cas d'épuisement de la mémoire.

Pour nos logiciels nous emploierons de même une classe et non un pointeur pour les chaînes qui doivent être copiées ou modifiées. Nous pourrons utiliser les `string` de la librairie standard ou les `rope` de la STL.

7.6.3 Copie des objets

La copie d'objets consiste à créer un *nouvel* objet ayant le même état abstrait qu'un objet existant. Elle s'effectue donc par un constructeur appelé *constructeur de copie* qui prend en paramètre une référence (souvent constante) à l'objet existant. Le problème est donc similaire à celui de l'affectation, bien qu'ici un nouvel objet doive être créé alors qu'il existait déjà pour l'affectation.

Comme dans le cas de l'affectation, une copie par défaut peut être générée par le compilateur quand tous les membres de l'objet admettent eux-mêmes une copie (explicite ou par défaut). La copie par défaut se fait en copiant membre à membre les composants, et elle est admissible dans les mêmes cas que l'affectation membre à membre.

Cette analogie conduit à considérer de manière conjointe les stratégies d'affectation et de copie dans une même classe.
Exemple :

```
Chaîne::Chaîne(const Chaîne& s):
  p(std::strcpy(new char[std::strlen(s.p)+1],s.p)){
}
```

Nous devons nous souvenir que les constructeurs de copies sont utilisés aussi pour copier les valeurs des arguments effectifs lors de l'appel des fonctions. Ainsi le constructeur de copie de `Chaîne` est appelé pour tout argument de type `Chaîne`

et provoque une nouvelle allocation mémoire et une recopie des caractères de la chaîne.

Il est impossible de passer par valeur l'argument du constructeur de copie car cela provoquerait une boucle de programme.

Règle 7.9 Les arguments du constructeur de copie et de l'affectation par copie doivent toujours être passés par références constantes

La complexité ou le temps nécessaire à une copie ne dépend pas toujours de la taille de l'objet, mais est déterminée par le constructeur de copie. Certaines classes peuvent en partageant les ressources permettre une copie[11] extrêmement rapide de gros objets voir §7.6.5.

La copie des `Chaînes` n'est pas seulement la copie du pointeur, mais implique une allocation mémoire et une recopie des caractères. Pour éviter cette copie coûteuse nous passerons les arguments de type `Chaîne` par références constantes, quand les chaînes ne doivent pas être changées par la procédure et que leur durée de vie excède celle de la procédure appelée.

Rec. 5.8 Évitez les recopies inutiles d'objets quand la copie est coûteuse.

En remplaçant la copie par une référence nous créons un partage temporaire de l'objet. C'est sans risque tant que *l'objet est constant* et que l'on ne stocke pas une référence ou un pointeur sur l'objet. Dans le cas contraire si la copie doit être évitée il nous faudra avoir recours aux techniques de la section §7.6.5.

7.6.4 Classe sans copie

On ne peut pas copier les objets de certaines classes, C'est le cas des objets qui ne peuvent avoir qu'une seule instance. Ils seront identifiés par une clé unique et souvent gérés au niveau de la classe par une liste statique d'instances.

Quand la copie est interdite on doit renoncer au passage par valeur des paramètres, ainsi qu'au retour de la valeur d'un objet par une fonction, qui tous deux impliquent une copie.

Pour interdire la copie et empêcher le compilateur de définir une copie par défaut on déclarera dans la partie privée de la classe le constructeur de copie et l'affectation. Il ne sera pas nécessaire de les définir puisqu'ils ne seront jamais utilisés.

Rec. 5.10 Quand les objets d'une classe ne doivent jamais être copiés, alors le constructeur de copie et l'affectation par copie seront déclarés `private` et ne seront pas implémentés.

[11] Il s'agit en fait d'une copie fictive, rien n'est réellement copié !

Si l'on interdit la copie dans une classe, non seulement il sera impossible d'avoir simultanément deux copies de l'objet, mais il sera aussi impossible de transférer un objet en le recopiant d'abord et en supprimant le vieil exemplaire ensuite. Comme cette méthode est utilisée par les conteneurs de la STL pour déplacer leurs objets quand les conteneurs sont devenus trop petits, les objets sans copie ne peuvent pas être placés directement dans un conteneur. [12]

La STL exige que les types d'objets placés dans un conteneur (*Container*) soient affectables (*Assignable*) ce qui implique qu'ils possèdent à la fois une copie et une affectation[13].

7.6.5 Partage de composants membres

Dans certains cas nous préférerons ne pas copier entièrement l'objet, mais laisser plusieurs clients en partager l'accès. C'est en particulier le cas quand, dans le domaine de l'application, l'objet est effectivement partagé, mais ce pourra aussi être adopté quand deux objets possèdent des membres constants dans le même état.

Les cas de partages effectifs dans le domaine de l'application sont nombreux : nous pouvons penser à des personnes qui partagent un bureau, un projet, la gestion d'un budget, ou encore à des systèmes d'exploitations qui partagent l'accès à des machines, à des système de fichiers, la gestion des postes de travail, etc.

Dans le cas où un client est l'unique propriétaire du membre et donc que sa durée de vie est plus grande que la durée de vie de l'objet à partager, il est aisé de confier la responsabilité de la création et destruction du membre à son propriétaire et de donner aux autres objets une référence, éventuellement constante à l'objet partagé. Ce sont les cas que l'on nomme *composition* en conception objet.

Un cas intermédiaire est celui d'un objet n'ayant qu'un seul propriétaire à une période donnée mais dont la propriété peut être transférée, et donc qui a une durée de vie indépendante de celle de son propriétaire. De tels exemple abondent dans le monde sensible et il sera utile pour les programmer d'avoir une classe de pointeurs qui sache gérer ces transferts de propriétés. La bibliothèque standard fournit dans l'en-tête `<memory>` une classe de *pointeurs intelligents* `auto_ptr` qui prend le contrôle de l'objet et en vérifie la destruction quand le pointeur est détruit. Les pointeurs `auto_ptr` permettent aussi le transfert de propriété avec un opérateur `=`.

La bibliothèque `BOOST` offre les `scoped_ptr` et `scoped_array` qui sont adaptés à un propriétaire unique et non modifiable.

Dans le cas le plus général, on ne peut pas assurer une telle exclusivité : l'ob-

[12] Les objets sans copie peuvent cependant être *empaquetés* dans des objets *enveloppe* qui eux disposent d'une copie et peuvent être placés dans les conteneurs.

[13] et une opération d'échange `swap`

Programme 7.11 Gestion partagée d'objets

```
class A{
public:
   A(const C &c):pc(new PartageC(c)){}          //Nouvel objet
   A(const A& x){pc=x.pc; pc->nbref++;} //partage d'un objet
   A& operator = ( A& x);               //affectation avec partage
   ~A(){if(--(pc->nbref)==0) delete pc;}//détruit si plus utilisé
....
private:
  struct PartageC {
    int nbref;
    C partage;
    PartageC(const C& c)nbref(1),C(c){};
  };
  PartageC* pc;
}
A& C::operator = ( A& x){
  if (&x !=this){        //sauf cas x=x
    if(--(pc->nbref)==0) delete pc;
    pc=x.pc;
    ++(pc->nbref);
   }
};
```

jet est réellement partagé. On a alors une *agrégation* simple et la destruction de l'objet partagé est plus délicate puisqu'il ne faut libérer une place mémoire que si l'on est sûr que l'objet n'a plus de client.

Nous pourrons alors employer un compteur de références qui permet de déterminer le moment où une zone peut être libérée. Ce compteur est réalisé grâce à un nombre entier, partagé lui aussi par les utilisateurs de l'objet. Il est incrémenté à chaque construction et décrémenté à chaque destruction.[14]

Nous donnons dans le programme 7.11 un schéma de programme de gestion partagée d'objets de type `C`, Le contrôle se fait par un compteur de nombres d'instance `nbref` qui est ici extérieur à l'objet `C`.

Un programme plus complet qui emploie des types Patrons est donné dans le programme 7.12 page 166.Il utilise une gestion *intrusive*, ce qui signifie que le nombre de références à l'objet est porté par l'objet lui-même, ce qui oblige

[14]Cette technique est très délicate quand les objets ne sont pas constants et que plusieurs *threads* (ou processus légers) peuvent y accéder ; les compteurs de références devraient alors être protégés par un sémaphore.

à préparer l'objet pour le partage, mais on gagnera une meilleure efficacité et la possibilité de conversion implicite des objets pointeurs qui reflète la conversion des pointeurs.

Dans les affectations il est nécessaire de prévoir le cas `x=x` pour ne pas détruire alors l'objet que l'on reçoit.

Règle 5.12 L'opérateur d'affectation doit être protégé contre la destruction d'un objet qui est affecté à lui-même.

La bibliothèque `BOOST` offre les `shared_ptr` et `shared_array` qui utilisent aussi la technique du comptage de référence pour permettre le partage de propriété et la destruction d'un objet ou d'un tableau.

7.6.6 Constructeurs de conversion

Un constructeur d'une classe qui accepte pour seul argument un objet d'un type différent, permet de convertir ce type vers la classe. Nous pouvons ainsi définir des constructeurs de conversion qui seront employés aussi bien pour créer de nouveaux objets en convertissant un objet d'une autre classe, que pour convertir les arguments effectifs d'une fonction.
Exemple :

```
Chaîne::Chaîne( const char* s):
   p(std::strcpy(new char[std::strlen(s)+1],s)){
}
```

Ce constructeur permet d'utiliser des `const char *` là où des arguments `Chaîne` sont attendus. Les constructeurs de conversion peuvent permettre de diminuer le nombre de méthodes surchargées nécessaires dans une classe, mais en sacrifiant la clarté du code et ses possibilités d'évolution.

Ainsi pour les chaînes nous avons défini une unique méthode de nom `ajouter` :

```
void Chaîne::ajouter( const Chaîne& s);
```

Avec le constructeur de conversion nous pouvons effectuer l'appel :

```
Chaîne s("un");
s.ajouter("et deux");
```

Le littéral chaîne sera converti en type `Chaîne` par le constructeur de conversion avant l'appel de la fonction `ajouter`. Il n'est pas nécessaire de définir une méthode `ajouter` qui ajoute un `const char *` : le constructeur de conversion permet d'utiliser la fonction déjà définie.

Bien entendu cette utilisation n'est possible que quand les arguments sont passés par copie ou par référence constante : le constructeur de copie ne serait d'aucune aide pour un argument passé comme référence non constante, mais nous avons convenu de ne pas utiliser de tels arguments dans des fonctions membres.

Notons cependant qu'ici, l'utilisation du constructeur de copie implique une recopie *inutile* de la chaîne dans une zone temporaire, que nous éviterons en surchargeant la fonction.

Si nous définissons maintenant un constructeur d'une chaîne vide de longueur arbitraire :

```
Chaîne::Chaîne( size_t n):
     p(new char[n])
{
  std::memset(p,'\0',n);
}
```

Nous définissons, là encore, un constructeur pouvant effectuer une conversion implicite, et l'erreur de l'utilisateur distrait qui écrit `s.ajout(1515)` au lieu d'un `s.ajout("1515")` ne sera pas détectée.

Ces erreurs seront évitées en utilisant le mot-clé `explicit` qui indique que le constructeur ne peut pas être utilisé pour une conversion implicite, mais seulement pour une conversion explicite.

Toutes les conversions implicites rendent le code peu lisible : pour savoir quelle fonction est appelée il faut examiner toutes les possibilités de conversion. Pour cela il faut examiner tout les en-têtes accessibles pour y chercher les constructeurs et les opérateurs de conversions implicites qui peuvent être appelés.

L'expérience montre que ces appels implicites sont habituellement oubliés dans la phase de mise au point et qu'ils sont donc une source d'erreurs persistantes.

Rec. 7.18 Les constructeurs à un argument doivent être déclarés `explicit`.

Nous définirons donc le constructeur des `Chaînes` par :

```
class Chaîne{
  public:
    explicit Chaîne( const char* s);
    explicit Chaîne(size_t n);
.....
};
```

Nous ne pourrons pas utiliser *de manière transparente* les pointeurs à la place des chaînes, mais nous pourrons toujours écrire :

```
Chaîne s("un");
s.ajouter(Chaîne("et deux"));
```

Ceci est d'une plus grande lisibilité et aidera éventuellement au débogage.

7.6.7 Opérateurs de conversion

L'utilisation des constructeurs de conversion permet de définir une conversion à l'entrée d'une classe. Elle ne permet pas de résoudre les conversions vers des types qui ne sont pas des classes, ou qui sont des classes que nous ne pouvons étendre (les classes de bibliothèque par exemple). Par exemple nous avons vu comment convertir les `const char *` en objets `Chaîne`, mais nous ne pouvons pas employer la même technique pour convertir les `Chaîne`s en `const char *`.

Le paragraphe §7.5.4 a déjà envisagé l'utilisation d'un accesseur :

```
const char * Chaîne::c_str(){ return p;}
```

que nous pouvons considérer comme un opérateur de conversion explicite.

Nous l'utilisons ainsi :

```
Chaîne s(" explicite! explicite!
est-ce que j'ai une gueule d'explicite!");"
cout << s.c_str()() << endl;
```

Pour éviter l'apparente lourdeur syntaxique de type de conversion, **C**++ permet d'effectuer les conversions de manière implicite avec l'opérateur :

```
operator const char * (){ return pM;}
```

Nous pouvons alors convertir une `Chaîne` en `const char*` soit par l'appel explicite `(const char*) s`, soit de manière implicite quand une méthode avec un argument formel de type `Chaîne` reçoit une valeur de type `const char*`. Nous pouvons alors écrire simplement :

```
Chaîne s("l'implicite peut-il être annoncé?");
cout << s << endl;
```

sans avoir surchargé `operator`<< pour la sortie des `Chaînes`.

Le compilateur ne fait jamais deux conversions implicites de suite : il n'y a pas transitivité des conversions implicites.

Comme les constructeurs de conversion implicites, les opérateurs de conversion implicites sont de fréquentes causes d'erreur. Ils induisent des appels de fonctions surchargées qui ne sont pas souhaités et rendent la maintenance délicate en *cachant* le code et en le *délocalisant*[15].

[15] voir aussi §2.9.1 et recommandation 6.1 page 45

Dans le cas particulier d'un retour de pointeur comme nous l'avons envisagé ci-dessus, la recommandation 10.2 page 150 nous demande de signaler à l'utilisateur la durée de validité du pointeur. C'est bien sûr incompatible avec l'*implicite*.

Malgré la simplicité apparente des conversions implicites nous leur préférerons les conversions explicites qui sont plus sûres.

Rec. 7.19 N'utilisez pas les fonctions de conversion implicites.

7.6.8 Surcharge des opérateurs prédéfinis.

Les classes nous permettent en particulier de définir de nouveaux types abstraits sur lesquels nous définissons des opérations. Dans certains cas ces types ont suffisamment d'analogie avec les types prédéfinis, pour que l'utilisation des opérateurs arithmétiques facilite grandement leur utilisation. Le langage **C**++ permet de surcharger les opérateurs arithmétiques pour permettre l'utilisation de ces opérateurs pour des types définis par l'utilisateur.
Exemple :

```
class Complex {
public:
 Complex( double r , double i ):reM(r),imM(i){}
 explicit Complex( double r=0.):reM(r),imM(0){}
.....
private:
  double reM;
  double imM;
};
```

Nous pouvons vouloir définir sur les objets de type `Complexe` les opérations arithmétiques usuelles et, pour prendre le cas de l'addition, donner un sens aux expressions suivantes :

```
Complex a(1.,2.);
Complex b(1.,1.);
Complex c; //initialisation à Complex(0.,0.)
c=a+b;  // (1)
c=a+Complex(1.) // (2)
c=a+1.; // (3)
c=c+1;  // (4)
c=1.+a; // (5)
c+=b;   // (6)
```

En effet les instructions figurant à droite de l'affectation sont interprétées par le compilateur comme des appels de fonctions. Pour résoudre l'expression (1) le compilateur recherchera une définition de la méthode : [16]

```
Complex Complex::operator + ( Complex)const;
```

L'expression (1) sera alors traduite en :

```
c=  a.operator + (b);
```

Le compilateur recherchera aussi une fonction globale :

```
Complex operator + ( Complex, Complex);
```

Pour traduire l'expression en :

```
c=operator+(a, b);
```

Il revient au programmeur de choisir entre ces deux solutions qui, nous le verrons, présentent des différences.

L'utilisation d'une méthode pour définir les opérateurs sera préférée à une fonction externe quand cela est possible ; nous considérerons donc tout opérateur comme l'application d'une méthode sur son premier argument et nous écrirons :

```
Complex Complex::operator + ( Complex c)const{
  return Complex( reM +c.reM, imM + c.imM);
}
```

Cette définition nous suffira à résoudre l'appel (1) ainsi que l'appel (2) ou la conversion est explicite.

L'application de cette méthode à (3) demanderait une conversion, et la conversion implicite est empêchée par le mot clé `explicit`. (cf.§7.6.7 et recommandations 15.17 et 6.1). Si nous désirons permettre l'emploi direct de (3) nous écrirons donc :

```
Complex Complex::operator + ( double c)const{
  return Complex( reM +c.reM, imM );
}
```

Pour (4) nous n'avons rien à rajouter car la conversion d'un entier en `double` est implicite.

Remarquons que même si une conversion implicite de `double` en `complexe` avait permis de résoudre (1) et (3) avec la même méthode, nous aurions quand

[16] L' argument est ici passé par copie, il pourrait aussi l'être par référence constante.

même eu besoin d'une méthode distincte pour (4) car il ne peut pas y avoir enchaînement de deux conversions implicites (§2.9.1).

Par contre (5) ne peut être résolu par une méthode de classe puisque son premier opérande n'est pas un objet de classe. Nous pouvons indiquer au compilateur que nous considérons le + comme commutatif en définissant la fonction externe :

```
inline Complex operator + ( double r, Complex c){
    return c.operator+(r);
}
```

Cette fonction n'a pas à être déclarée amie, car elle n'utilise que le protocole public des complexes, elle est donc préférable à une manipulation directe des membres de l'objet c.

L'appel (6) peut être résolu par la méthode :

```
Complex& Complex::operator += ( Complex x){
  re+=x.re;
  im+=x.im;
  return *this;
}
```

Cette définition indépendante du + et du += permet ici d'optimiser l'opérateur +=, quand les méthodes sont compliquées on préférera souvent relier les deux méthodes en écrivant par exemple :

```
inline Complex Complex::operator + ( Complex c)const{
 return Complex(*this)+=c;
}
```

Opérations entre types hétérogènes.

L'addition d'un complexe et d'un réel nous a donné un exemple d'opération entre deux données de types différents. Nous avons choisi d'interpréter de la même manière la somme d'un réel et d'un complexe quelle que soit la position des opérandes, en préservant la commutativité usuelle de l'opérateur +.

Pour les fonctions externes, ou à types hétérogènes la notation x*a pour un objet de classe X représente généralement une *action* de la valeur a sur l'objet x. Nous appliquerons une méthode de classe :

```
X X::operator * (const A a&);
```

Le même type d'opération est parfois écrit de `a*x`, en particulier dans les notations mathématiques des espaces vectoriels ; dans ce cas on aura recours à une fonction globale comme nous l'avons fait pour la somme d'un réel et d'un complexe, pour spécifier que `a*x` est une application d'une fonction de paramètre `a` à l'objet `x` et non une opération sur `a`, et qu'il convient donc d'inverser les arguments.

```
inline X operator * (const A a& , const X& x){
  return x.operator *(a);
}
```

Certains programmeurs usent des constructeurs de conversion et des opérateurs de conversion, pour résoudre les opérateurs agissant sur des objets de deux classes différentes. Mais il est souvent difficile d'estimer si une conversion implicite correspond à une action désirée ou involontaire.

De plus dans le cas où deux conversions sont possibles le compilateur ne peut pas résoudre l'ambiguïté, et la signale par un message d'erreur, quand bien même les deux conversions auraient produit des résultats identiques.

Nous accepterons donc d'écrire quelques lignes de code supplémentaires pour guider le compilateur vers la méthode adéquate et nous restreindrons les cas de conversion aux types prédéfinis et aux héritages publics.

Définition d'une famille d'opérateurs

Quand nous définissons des opérateurs il est important de s'assurer que nous ne contredisons pas les propriétés qu'attribue l'usage à ces opérateurs.

Il en est ainsi pour les opérateurs `+` et `*` qui sont réputés commutatifs et associatifs quand les deux opérandes sont du même type, , règle à laquelle on ne dérogera que dans des cas exceptionnels.

De même l'utilisateur attend que `x+=y;` ait le même effet que `x=x+y;` ou encore que `x<=y` soit identique à `!(x>y)` et on tâchera de se conformer à de telles habitudes.

Rec. 7.15 Si vous surchargez des opérations d'une famille d'opérateurs apparentés vous devez surcharger toute la famille et préserver les invariants qui existent pour les types prédéfinis.

L'inclusion de l'en-tête de la STL `<utility>` demande à la *Standard Template Library* (Chap. §10) de générer automatiquement les opérateurs relationnels manquants en fonction de ceux qui sont présents. Ce mécanisme est détaillé §10.3.1. Pour cela la STL suppose que nous voulons réaliser une relation d'ordre total.

Ceci nous permettra donc de satisfaire à la recommandation 7.15 à moindre frais pour ces opérateurs. En effet il suffit de définir `<` et `==`, pour générer `!=`, `<=`, `>=`, `>` de manière à réaliser un ordre total.

Le revers de la génération d'opérateur par la STL est que nous devrons nous garder de l'apparition non souhaitée d'opérateurs qui n'auraient pas le sens voulu.

Dans l'exemple des chaînes de caractères qui suit nous donnons à l'opérateur `<` le sens d'une comparaison lexicographique. C'est un ordre total et nous pouvons laisser la STL générer les opérateurs manquants. Si nous préférons l'ordre *préfixe* qui est un ordre partiel, les opérateurs relationels patrons de la STL génèrerait un ordre total ce qui ne convient pas.

Si nous incluons l'en-tête `<utility>`(en fait le fichier responsable inclus est <stl_relops.h>), le compilateur suppose que nous désirons des ordres totaux, et il génère les opérateurs manquants en fonction de cette hypothèse.

Je ne peux que citer le début l'introduction de l'en-tête de <stl_relops.h> dans la version 3 de <libstdc++> : « The rel_ops operators cannot be made to play nice. Don't use them.» *you have been warned* – on vous a prévenu !

Dans le cas ou le comportement des opérateurs s'écarte par trop de celui des types prédéfinis, il faut examiner si vraiment un opérateur arithmétique est à préférer à une autre fonction membre.

Usage non standard des opérateurs.

Les opérateurs peuvent être étendus à de nombreux types abstraits, quand bien même ils ne correspondent pas aux types arithmétiques usuels. Par exemple, nous pouvons pour le type chaîne définir :

```
class Chaîne {
public:
.....
  Chaîne& operator+= (const Chaîne& s);
    //this = concaténation de this et s
  bool operator <= (const Chaîne& s)const;// ordre
  bool operator < (const Chaîne& s)const; // lexicographique
  bool operator >= (const Chaîne& s)const {return !operator<(s);}
  bool operator > (const Chaîne& s)const {return !operator<=(s);}
  bool operator == (const Chaîne& s)const;
  bool operator != (const Chaîne& s)const {return !operator==(s);}
  char& operator [] (unsigned int i){return pM[i]};
  char operator [] (unsigned int i)const {return pM[i]};
  Chaîne operator+ ( const Chaîne& t) const; // concaténation
```

```
private:
  char * pM;
};
```

L'opérateur `[ ]` pour lequel un traitement des erreurs d'indice serait approprié permet de définir un opérateur d'indexation dans une classe. Pour un objet constant il rend la valeur du $(i+1)^{eme}$ élément de l'objet, en revanche pour un objet non constant il rend une *référence* au $(i+1)^{eme}$ élément, ce qui permet de modifier cet élément[17]. Dans tous ces cas *non standards* il convient de vérifier que les propriétés usuelles des opérateurs sont conservées, et sinon, de préférer des fonctions membres ordinaires.

7.7 Structures

Une *structure* est par définition une classe dont les membres sont publics par défaut. Toutes les autres caractéristiques sont identiques à celles des classes.

Les structures comprennent *mais ne sont pas limitées* aux agrégations de champs publics comme le sont les structures **C**.

Nous avons convenu de toujours utiliser dans les définitions de classe les mots clés `public` et `private`, nous pouvons donc choisir de réserver le mot clé `struct` pour les classe dont les champs sont publics. Ces classes sont généralement des classes *concrètes* qui servent de blocs de base, et qui ne sont atteintes que par des classes abstraites qui fournissent l'interface public.

Style A.14 Le mot-clé `struct` sera utilisé seulement pour les structures dans le style du langage **C**.

[17] voir aussi §10.1 l'indexation dans la classe `vector` et §11.3 le cas de `string`

Programme 7.12 Classe `IRef`, gestion intrusive du partage (1).

```
template<class T, class U>
T implicit_cast( const U& x ) {
   return x;
}
typedef unsigned int RefCountType;
template <typename U> class IRef;

class IRefable {
   private:
   mutable RefCountType refC_;
   protected:
      IRefable(RefCountType x = 0) : refC_(x) {}
   public:
      RefCountType use_count()const{ return refC_;}
      void incref() const { ++refC_; }
      bool decref() const { return --refC_==0;}
};

template <class T>
class IRef{
   public:
      typedef T WrappedType;
      IRef() : ptr(0){}
// The following method is OK with IRef r(new ...) but problematic
//  if the adress of the pointer is yet stored, in any case the pointer is
// owned by IRef and will be deleted after use.
      explicit IRef(T* p=0) : ptr(p) {incref();}
      explicit IRef(const T& t): ptr(new T(t)){incref();}
      // we need a friend clause to access ir.ptr
      template <typename U>  IRef(const IRef<U>& ir):
         ptr(ir.ptr) {incref();}
      IRef(const IRef& other): ptr(other.ptr) {incref();}
      ~IRef(){ dispose();  }

      template <typename U> IRef<T>& operator=(const IRef<U>& other){
         share(other.ptr);
         return *this;
      }
       IRef& operator=(const IRef& other){
         share(other.ptr);
         return *this;
      }
      void reset(T* p=0) {
         if (ptr==p) return;
         dispose();
         ptr=p;
         incref();
      }
```

Programme 7.13 Classe `IRef`, gestion intrusive du partage.(2)

```
      T* get() const { return ptr; }
      T& operator*() const { return *ptr;}
      T* operator->() const { return ptr; }
     static void swap( IRef<T>& x, IRef<T>& y ){
      T* tmp = x.ptr;
      x.ptr = y.ptr;
      y.ptr = tmp;
   }
      RefCountType use_count() const {
         if (ptr==0) {
            return 0;
            }else{
               return ptr->use_count();
            }
      }
   protected:
      T* ptr;
      // the const can seem strange but an other Iref can increment
      // the count through the friend clause, so the number of ref
      // is not relevant for the const concept.
      void incref() const {if (ptr!=0) ptr->incref();}
      template <typename U> friend class IRef;
       void dispose(){if (ptr->decref()) checked_delete(ptr); ptr=0;}
      void share(T* ptrn){
         if( ptr != ptrn){
            ptrn->incref(); // done before dispose() in case ptrn
                            // transitively dependent on ptr
            dispose();
            ptr=ptrn;
         }
      } //share
};
```

Chapitre 8

Héritage

8.1 Spécialisation d'une classe

Considérons le cas du parc de véhicules d'une entreprise. Les services offerts par ces véhicules sont variés. Certains services sont offerts par tous les véhicules, tels que se déplacer (ou `rouler`). Certaines méthodes, bien que communes à certains véhicules ne sont pas applicables à tous. Par exemple `ravitailler` n'a pas de sens pour une remorque, la `charge utile` n'est connue que pour les remorques et les camions, etc.

Si nous essayons de regrouper en classes les objets qui ont un protocole commun, nous obtenons une multitude de classes avec un grand nombre de méthodes identiques.

Les objets ont des protocoles qui ont une partie commune et une partie distincte.

Une caractéristique essentielle des langages objets est d'offrir un support syntaxique pour l'*héritage* ; c'est-à-dire de permettre de factoriser les protocoles communs des objets.

L'analyse donnera donc un diagramme de classe de l'application `Véhicules` du type du diagramme 8.1

Nous pouvons alors créer des véhicules par :

```
Voiture deuche("2CV", "130X13", 150000, 450, 25., "1515AA22");
Scooter vespa("ciao", "105X7", 15, 49, 4.);
```

Les méthodes pourront s'écrire :

```
void Véhicule::rouler(int nbkms){
  kilométrageM+=nbkms;
}
ostream& Véhicule::décrire(ostream& os){
```

Programme 8.1 En-tête C++ des classes de l'application `Véhicules`

```
class Véhicule{
public:
  Véhicule(Modèle m, Pneus p,int kmIni);
  ostream& décrire(ostream& os);
  void rouler(int nbkms);
  Modèle modèle()const;
  int kilométrage()const;
  Pneus pneus()const;
private:
  Modèle modM;
  int kilométrageM;
  Pneus pneusM;
};

class VéhiculeMotorisé: public Véhicule{
public:
  VéhiculeMotorisé(Modèle m , Pneus p, int kmIni,
                   int cyl, float capacitéReservoir);
    // retour: consommation possible
  ostream& décrire(ostream& os)const;
  bool rouler(int nbkms, float consommation);
  void ravitailler( float qté);
  float capacité()const; //capacité du réservoir
  float niveau()const;   //quantité restante d'essence
  int cylindrée()const;
private:
  int cylindréeM;
  Carburant carburantM;
  float capacitéM;
  float niveauM;
};

class Voiture: public VéhiculeMotorisé{
  Voiture(Modèle m , Pneus p, , int kmIni, int cyl
          float capacitéReservoir, string immatN );
  ostream& décrire(ostream& os)const;
  string immatriculation()const;
  void immatriculation(string immatN);
private:
  string immatriculationM;
}
```

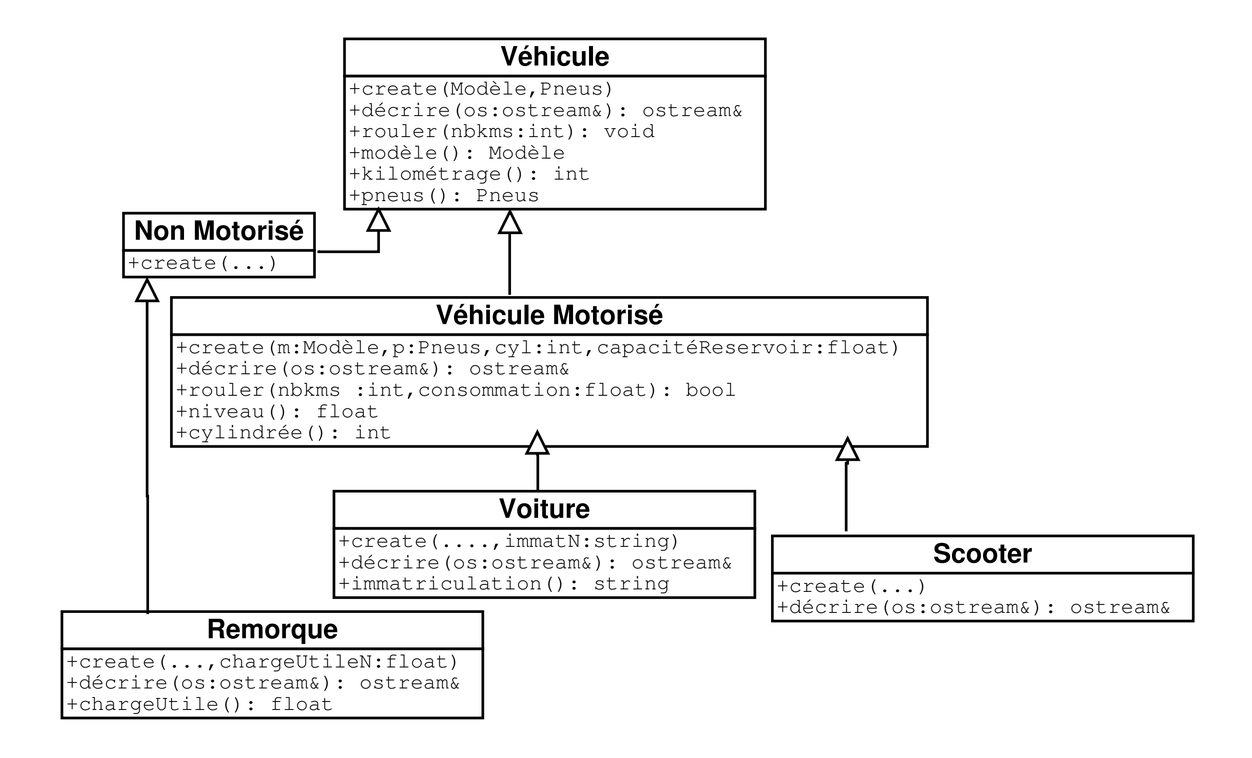


FIG. 8.1 – Diagramme UML de l'application Véhicules

```
  os << "Modèle: "<<modèle() << "Pneus: " << pneus() <<endl;
  os << kilomètrage() <<"km "<< endl;
  return os;
}

bool Véhicule::rouler(int nbkms, float consommation){
  kilométrageM += nbkms;
  return (niveauM -= consommation) > 0;
}
```

Le mot clé `public` indique que tous les membres publics de `Véhicule` sont aussi des membres publics de `VéhiculeMotorisé`. Un `VéhiculeMotorisé` aura donc comme membres publics

les méthodes de `Véhicule` **:**
: `décrire(ostream&) const`, `rouler(int)`, `modèle() const`, `kilométrage() const`, et `pneus() const`

les méthodes de `VéhiculeMotorisé` **:**
: `décrire(ostream&) const`, `rouler(int ,float )`, `capacité() const`, `niveau() const`, `cylindrée() const`.

Il aura comme membres privés cylindréeM, et niveauM, en revanche les membres modM, kilométrageM, pneusM ne sont pas des membres privés et ne sont donc pas accessible dans le corps des méthodes définies dans Véhicule-Motorisé. Nous devrons donc écrire la méthode rouler(int,float) sans accéder à kilométrageM :

```
bool VéhiculeMotorisé::rouler(int nbkms, float consommation){
  Véhicule::rouler(nbkms); // méthode de la classe mère
  return (niveauM -= consommation) > 0;
}
```

On utilise ainsi la méthode rouler de la classe mère Véhicule, qui est publique donc accessibles depuis tout objet Véhicule. Cependant la présence de rouler(int) dans le protocole public de VéhiculeMotorisé n'est généralement pas souhaitée : la section §8.2.2 indique comment l'éviter.

Deux méthodes publiques de même nom décrire(ostream&) const sont définies dans le protocole public de VéhiculeMotorisé. Pour un objet de la classe VéhiculeMotorisé, la méthode de sa classe *cache* la méthode d'une classe ancêtre, nous sommes donc assurés que la méthode décrire(ostream&)const adéquate sera appelée. Cependant il est toujours possible d'accéder à la méthode de la classe mère en la qualifiant par le nom de sa classe.
Exemple :

```
ostream& VéhiculeMotorisé::décrire(ostream& os){
  Véhicule::décrire(os)
  os << "Cylindrée: "<<cylindrée();
  os << "Niveau" <<niveau() <<"/"<<capacité()<<" l."<<endl;
  return os;
}
```

Le cas des constructeurs est particulier. Un constructeur n'est appelé que lors de la définition d'un objet. Le constructeur de Véhicule est donc appelé pour la construction d'un Véhicule alors que les objets de classe VéhiculeMotorisé utiliseront leurs propres constructeurs. Mais puisqu'un véhicule motorisé est d'abord un véhicule, la construction d'un VéhiculeMotorisé doit commencer par celle d'un Véhicule. La syntaxe des constructeurs est expliquée §8.2.6.

La destruction d'un VéhiculeMotorisé, quant à elle, devra se terminer par la destruction du Véhicule.

La classe Véhicule est une *classe de base* de la *classe dérivée* VéhiculeMotorisé, nous parlerons aussi de classe *mère* et de classe *fille*.

Si nous avons par ailleurs une classe Personne et que nous savons que chaque personne possède exactement un véhicule nous pourrons avoir un objet membre Véhicule à l'intérieur de Personne. Nous déclarerons donc :

```
class Personne{
public:
  Personne(.... ,Véhicule v);
  .....
private:
  véhicule véhiculeM;
 ....
};
```

En effet si nous initialisons une personne avec :

```
Personne toto(...., Scooter(...));
```

l'appel du constructeur va convertir l'argument effectif qui est un `Scooter` en `Véhicule`. Cette conversion est possible car l'héritage est public, mais seuls seront pris en compte les champs de `VéhiculeMotorisé` qui sont dans la classe de base `Véhicule`. Il sera donc impossible de connaître le kilométrage ou le niveau d'essence.

Nous aurions pu choisir de conserver une référence au véhicule plutôt qu'une copie. Pour cela nous aurions déclaré :

```
  Personne(.... ,Véhicule& v);
  .....
private:
 véhicule& véhiculeM;
```

Dans le cas ou le véhicule n'existe pas toujours *avant* la création de la personne, et *jusqu'à* la destruction de la personne une reférence ne peut pas être utilisée et nous prendrons alors un pointeur :

```
  Personne(.... ,Véhicule* pv);
  .....
private:
  véhicule* pVéhicule;
```

Nous initialiserons `véhiculeM` ou `pVéhicule` à la référence ou à un pointeur sur le véhicule motorisé `maMob` par :

```
Scooter maMob(.....);
Personne toto(...., maMob);
```

ou respectivement :

```
Personne toto(...., &maMob);
```

Cette initialisation se ferait plus clairement avec une conversion explicite :

```
Personne toto(....,static_cast<Véhicule&>( maMob));
```

ou

```
Personne toto(....,static_cast<Véhicule*>( &maMob));
```

Règle de conversion : Quand l'héritage est déclaré *public*, les pointeurs et références sur un objet de la classe dérivée sont implicitement convertis dans la classe de base, chaque fois qu'une conversion de paramètre est nécessaire.

Il peut sembler alors que l'intégralité de la sous-classe de `Véhicule` utilisée est disponible, mais nous n'avons accès qu'au sous objet de type `Véhicule` et tout essai pour appeler avec cette référence ou ce pointeur les fonctions :

```
rouler(int , float), ravitailler(float), niveau(), cylindrée()
```

sera rejeté par le compilateur qui ne peut associer à `véhiculeM` (resp. `pVéhicule`) que des méthodes de sa classe déclarée c'est à dire `Véhicule` (resp. `Véhicule*`).

Il n'y a pas de conversion implicite d'un pointeur sur une classe de base vers un pointeur sur une classe fille, même si l'héritage est public.

La plus grande partie de ces conversions de la classe de base vers la classe dérivée seront rendues inutiles par l'emploi des fonctions virtuelles (cf §8.4) qui constituent la manière la plus adaptée pour traiter chaque sous classe de manière spécifique quand le type effectif de l'objet pointé n'est pas connu à la compilation.

Cependant si les fonctions virtuelles permettent de différencier les traitements selon le type de la sous-classe, elles ne résolvent pas le cas ou certaines sous classes ont besoin d'un traitement qui n'a pas de sens que pour elles.

Par exemple dans notre application la `cylindrée` n'a pas de sens pour un `VéhiculeNonMotorisé` et les fonctions virtuelles sont de peu d'aide pour différencier les protocoles.

Dans certains cas où l'inspection de l'objet pointé par `pVéhicule` peut nous permettre de savoir qu'il s'agit bien d'un véhicule motorisé, il peut être utile de retrouver l'objet initial. Nous pouvons alors effectuer une conversion explicite, elle est généralement peu sure car le compilateur ne peut pas vérifier que l'opérateur de transtypage est utilisé à bonne escient et que l'on accède réellement à un objet de la classe dérivée. Nous pourrons alors employer l'opérateur `dynamic_cast`.

La conversion :

```
VéhiculeMotorisé * pvm (
               dynamic_cast<VéhiculeMotorisé*>(pVéhicule));
```

est vérifiée dynamiquement à l'exécution, elle réussit si pVéhicule pointe effectivement sur un pVéhiculeMotorisé et rend 0 sinon. Il en est de même pour une référence, la conversion :

```
VéhiculeMotorisé & vm (
               dynamic_cast<VéhiculeMotorisé&>(véhiculeM));
```

réussit si pVéhicule est une référence à un pVéhiculeMotorisé si ce n'est pas le cas une exception bad_cast sera lancée.

8.2 Accès aux membres d'une classe de base.

Nous avons vu jusqu'à présent comment un héritage public transmettait les droits sur les objets membres. Nous pouvons choisir une autre *spécification d'accès* qui restreint les droits d'accès.

8.2.1 Interface protégé

L'appel à Véhicule::rouler(int) sur des objets de classe VéhiculeMotorisé est dangereux, en effet elle court-circuite la méthode rouler(int,float).

Elle reflète une erreur d'analyse, les automobiles ne sont pas des bicyclettes avec des possibilités supplémentaires, mais ces deux classes sont des spécialisations distinctes d'un même ancêtre.

Nous devons plutôt écrire :

```
class VéhiculeNonMotorisé: public Véhicule{
public:
  VéhiculeNonMotorisé(Modèle m , Pneus p);
  bool rouler(int nbkms);
  ...........
};
```

Ainsi il est possible de supprimer rouler(int) de la partie publique de Véhicule. Nous devons maintenant nous demander comment modifier le kilométrage, comme celui-ci peut être changé dans des véhicules motorisés, comme dans des véhicules non motorisés, les méthodes des deux classes VéhiculeMotorisé et VéhiculeNonMotorisé doivent partager ce membre de la classe mère.

La classe d'accès *protégé* déclarée par le spécificateur protected regroupe les membres qui peuvent être vus dans les méthodes des classes dérivées comme dans les méthodes de la classe mère, mais que les utilisateur extérieurs ne peuvent pas appeler.

Ici nous déclarerons :

```
Class Véhicule{
public:
 .......
protected:
  int kilométrageM;
private:
  Modèle modM;
  Pneus pneusM;
};
```

Il est alors possible d'écrire :

```
bool VéhiculeMotorisé::rouler(int nbkms, float consommation){
  kilométrageM += nbkms;
  return (niveauM -= consommation) > 0;
}
```

ainsi que :

```
void VéhiculeNonMotorisé::rouler(int nbkms){
  kilométrageM += nbkms;
}
```

Cette technique est correcte mais elle implique que la responsabilité de la cohérence des données partagées est répartie entre toutes les classes héritières de la classe qui fournit la donnée protégée. C'est une contrainte acceptable dans le cas où nous fournissons un système à l'utilisateur final, et où l'équipe de programmation maîtrise la diffusion non seulement du code source mais aussi des modules objets.

En revanche si nous fournissons une librairie qui sert au développement d'autres produits, cette stratégie est très dangereuse. Tout utilisateur de la librairie peut créer une classe qui utilise les variables protégées de manière incontrôlée. Nous allons donner un exemple de ce qu'un programmeur malveillant, facétieux ou distrait peut ajouter à nos classes sans même que nous lui en ayons fourni les sources.

Dans la classe `Véhicule` nous supposons que le kilométrage est toujours positif et ne fait que croître. Mais un programmeur peut écrire :

```
class MachineAremonterLespace: public Véhicule{
public:
   ...............
  void marcheArriere(int nbkm){  kilométrageM -= nbkms;}
  void retourCaseDépart(){ kilométrageM =0;}
  void getLost(){ kilométrageM =INT_MIN;}
```

```
   ..........
};
```

Cette classe peut en altérant l'intégrité de notre classe de base provoquer des erreurs dans celle ci, que nous détecterons difficilement et, en tout état de cause, que nous ne pourrons pas corriger.

Remarquons tout d'abord que ces erreurs seraient, malgré tout, interceptées par une vérification d'invariant dans la classe de base, et ceci avant d'avoir pu provoquer des dommages importants. Mais s'il est utile d'identifier les erreurs, il est encore plus important de les éviter.

Il est donc souhaitable ici de laisser `kilométrageM` comme membre privé et n'offrir aux descendants qu'une méthode qui *protège* réellement l'accès :

```
Class Véhicule{
public:
  Véhicule(Modèle, Pneus);
  Modèle modèle();
  int kilométrage();
  Pneus pneus();
  ...........
protected:
 void rouler(int nbkms);
private:
  Modèle modM;
  int kilométrageM;
  Pneus pneusM;
};
```

8.2.2 Déclaration d'accès à une sur-classe

La syntaxe pour déclarer une sur-classe est :

```
class A: accès B {
déclaration de la classe
};
```

La spécification d'accès peut être `public`, `protected`, ou `private` et précise comment une classe transmet *par défaut* son protocole à la classe fille. Il est possible de préciser ultérieurement certaines expansions des droits pour des membres particuliers.(voir §8.2.3.)

Quand il n'y a pas de spécification d'accès l'accès est par défaut privé pour une `classe` et public pour une `struct`ure. Mais nous avons convenu dans la convention de style A.12 (p. 121) de toujours donner un spécificateur d'accès.

Nous résumons dans les paragraphes qui suivent les règles de la transmission des droits d'accès.

Héritage public

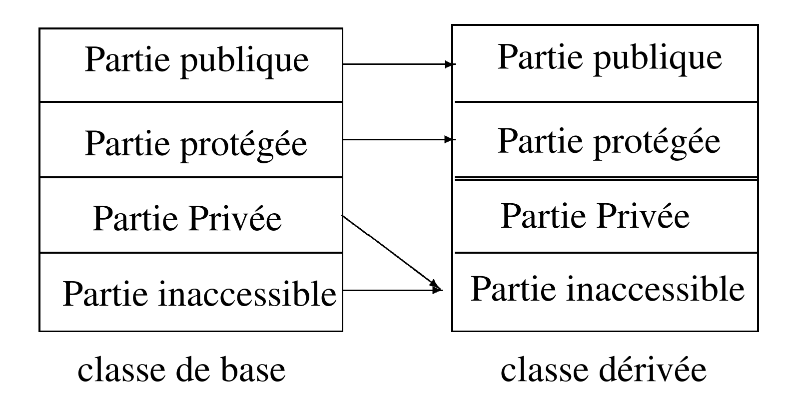


FIG. 8.2 – Transmission des droits d'accès dans un héritage public

Quand l'héritage est *public* les membres *publics* de la classe de base sont des membres publics de la classe dérivée . Les méthodes publiques de la classe de base font donc partie du protocole de la classe dérivée.

Les membres *privés* de la classe de base **ne sont pas des membres privés** des classes dérivées. Les fonctions membres des classes dérivées ne peuvent pas y accéder. Toute référence à une fonction privée d'une classe ne peut se faire que dans les membres de la classe ou dans les fonctions amies.

Les membres *protégés* sont des membres protégés des classes dérivées avec le spécificateur `public`. Les objets de la classe dérivée peuvent y accéder dans leurs méthodes, mais ne les intègrent pas à leur protocole public.

Un pointeur sur un objet de la classe dérivée peut être converti de manière implicite en un pointeur sur un objet de la classe de base, de même une référence à un objet de la classe dérivée peut être convertie en une référence à la classe de base.

Héritage protégé

Quand l'héritage est *protégé* (`protected`)les membres *publics* et les membres *protégés* de la classe de base sont des membres protégés des classes dérivées avec

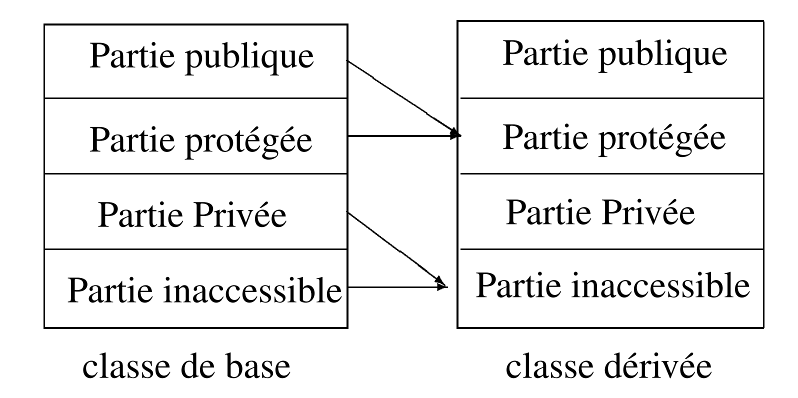


FIG. 8.3 – Transmission des droits d'accès dans un héritage protégé

le spécificateur d'accès `protected`, les méthodes publiques de la classe de base **ne font donc pas** partie du protocole public de la classe dérivée, mais les objets de la classe dérivée peuvent appeler de l'intérieur de leurs méthodes les membres publics ou protégés de la classe de base.

Les conversions implicites de pointeurs ou références de la classe dérivée vers la classe de base ne peuvent avoir lieu qu'à l'intérieur des fonctions membres de la classe dérivée ou de ses sous classes[1].

Héritage privé

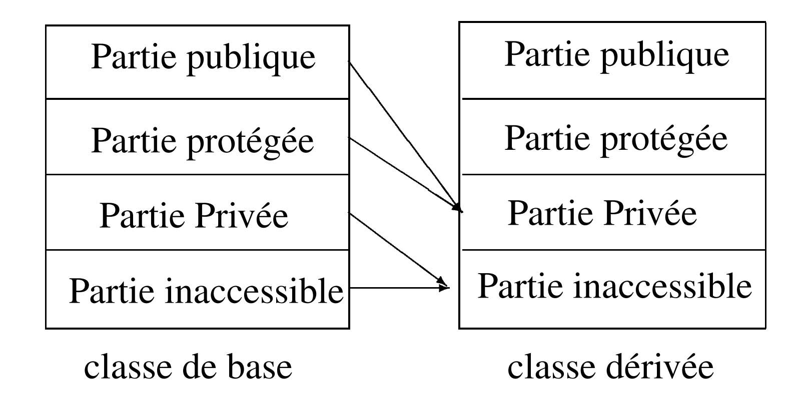


FIG. 8.4 – Transmission des droits d'accès dans un héritage privé

Quand l'héritage est privé, les méthodes *publiques* et *protégées* de la classe de base sont des méthodes *privées* de la classe fille. Elles ne sont donc pas intégrées

[1]Rappelons que les transtypages explicites de pointeurs sont toujours possibles

au protocole de celle-ci. Les conversions de pointeurs ou références de la classe dérivée vers la classe de base ne peuvent avoir lieu qu'à l'intérieur des fonctions membres de la classe dérivée.

Exemple d'héritage.

```
class B {
     public:
        int mi;
        static int si;  // membre statique
};
class D : private B {
};
class DD : public D {
        void f();
};
void DD::f() {
        mi = 3;        // erreur: mi est privé dans D
        si = 3;        // erreur: si est privé dans D
        B  b;
        b.mi = 3;      // ok b.mi est un membre public de b
        b.si = 3;      // ok b.si est B::si (statique public)
        B::si = 3;     // idem
        B* bp1 = this; // erreur : pas de conversion
                       // dans une classe de base privée
        B* bp2 = (B*)this;  // ok avec transtypage
        bp2->mi = 3;   // ok accès à un membre public de b
}
```

8.2.3 Définition d'un accès spécifique pour une variable

Il est possible de préciser l'accès à certaines variables héritées. Nous le ferons en utilisant la déclaration `using` déjà envisagée dans le contexte des espaces de nom au paragraphe 2.1.6. Elle servira ici à déclarer un synonyme d'un nom à l'intérieur d'une classe.

```
class A {
 public:
   int b, c;
 protected:
   void g();
```

```
 private:
   int a;
   void f(char);
};
 class B : private A {
   using A::f; // erreur: A::f(char) est inaccessible
 public:
   using B::b; // b est public dans B
   using A::g; // B::g est un synonyme public pour A::g
 protected:
   using B::c; // c est protégé dans B
 };
```

8.2.4 Fonction amie

Une fonction amie est une fonction qui n'est pas membre de la classe mais qui a le droit d'utiliser les membres protégés et privés de la classe.

La méthode amie ne fait pas partie de la classe et elle ne sera pas appelée comme une méthode de la classe. Elle ne sera pas appelée par un objet, sauf si elle fait partie d'une autre classe.

Exemple :

```
class Angle{
public:
  explicit Angle(double);
  friend float  sinus(const A&);
  friend istream& operator >> (istream & is , Angle & a);
private:
  float aM;

.....
};
float  sinus(const A& alpha){
  return std::sin(alpha.r);}
istream operator >> (istream & is,
                      Angle & a){
  return is >> a.aM;
}
```

Les fonctions amies propagent l'accès aux membres privés hors de la classe ; en *délocalisant* le contrôle elles peuvent nuire à la clarté des programmes et en

rendre la maintenance difficile. On n'utilisera donc des fonctions amies que dans le cas ou une méthode de classe ne peut pas convenir.

La fonction `sinus` devrait être remplacée par une méthode ou utiliser un accesseur.

La fonction d'entrée peut ne pas être déclarée comme amie en l'écrivant :

```
ostream operator >> (ostream & s , Angle & a){
  float x;
  s >> x;
  a=Angle(x);
  return s;
}
```

Ainsi la normalisation de l'angle est laissée au constructeur Angle(double) et à `Angle::operator =`.

Une autre méthode élégante pour contrôler la diffusion des noms est d'utiliser les *espaces de noms* (voir 2.1.6).

8.2.5 Classe amie

Une classe peut être déclarée amie d'une autre classe. La déclaration :

```
class A{
  friend class B;
private:
  int f(int);
  int a;
};
```

déclare la classe B comme amie de la classe A. Le membre `a` ou la fonction `f` peuvent alors être utilisés dans les fonctions membres de B.

L'accès aux membres amis ne se propage pas par transitivité. Autrement dit les amis des amis *ne sont pas* des amis.

Exemple :

```
class B {
  friend class C;
};
class C  {
  void g(A* p){
     p->a++;  // erreur : C n'est pas amie de A
  }
};
```

Les membres amis ne sont pas non plus hérités,

```
class D: public B   {
  int g(A* p, int i){
    return p->f(i);  // erreur : D n'est pas amie de A
  }                  // la fonction f n'est pas accessible
};
```

En revanche les amis d'une classe ont accès aux membres des sur-classes dans les mêmes conditions que les fonctions membres de la classe.

8.2.6 Constructeurs d'une classe dérivée

Pour construire un objet d'une classe dérivée il faut initialiser les objets membres de la classe de base et ceux de la classe dérivée.

L'ordre des initialisations est :

- La (ou les) classe (s) de base (dans l'ordre de déclaration).
- Les objets membres dans l'ordre de déclaration.
- Le corps de la fonction.

Si la classe de base est elle-même une classe dérivée, les appels aux constructeurs et destructeurs se propagent le long de la hiérarchie de classe.

La classe de base est toujours construite avant la classe dérivée, que cela soit par un appel explicite à un constructeurs, ou par un appel au constructeur par défaut (sans argument) .

Si le constructeurs par défaut n'existe pas, il est donc indispensable de construire explicitement la classe de base.

La construction explicite est dans tous les cas recommandée, même pour appeler explicitement un constructeur par défaut. Nous marquons là que l'appel est le fruit d'une intention et non d'un oubli.

Nous devrons être particulièrement attentif aux constructeurs de copie. Le constructeur de copie par défaut appelle les constructeurs de copie de ses classes de base et de ses objets membres. Si l'utilisateur choisit de définir un constructeur de copie explicite, il n'y a aucun appel par défaut et c'est la responsabilité de l'utilisateur de copier les classes de base et les objets membres.

8.2.7 Destruction d'une classe dérivée.

Lors d'une destruction d'un objet d'une classe dérivée, le destructeur de la classe est appelé, il est suivi des destructeurs des membres et enfin de celui de la classe de base. Bien entendu si la classe de base est elle-même une classe dérivée, les appels aux destructeurs se propagent le long de la hiérarchie de classe.

Contrairement aux constructeurs, les destructeurs des membres et des classes de base *sont toujours appelés implicitement*[2]. Pour que la destruction soit effectuée correctement il est donc nécessaire que le compilateur connaisse la classe de l'objet détruit. Pour une variable automatique le type de l'objet est toujours connu mais si on détruit un objet en mémoire libre par l'intermédiaire d' un pointeur d'une classe de base la destruction ne sera pas correctement effectuée à moins que le destructeur ne soit déclaré virtuel (*c.f.* §8.4.6).

8.2.8 Affectation dans une classe dérivée.

Puisqu'une fonction d'affectation est générée automatiquement par le compilateur quand elle est omise par le programmeur, il ne peut jamais y avoir héritage des affectations.

La fonction d'affectation par défaut commence par effectuer une affectation de la (ou des) classe(s) de base avant d'effectuer une affectation pour chaque membre.

Si nous définissons une fonction d'affectation il nous reviendra de copier la ou les classes de base.

8.3 Héritage multiple

8.3.1 Une autre conception de l'héritage.

Nous avons considéré jusqu'à maintenant le cas ou l'héritage correspond à une spécialisation de la classe mère, c'est à dire, quand une classe fille enrichit le protocole de la classe mère par de nouvelles méthodes. Ce cas est bien traduit dans les langages objets par un héritage public. Il correspond à la vue qu'a un zoologue quand il considère un être vivant : l'homme est une sorte de mammifère qui est lui même une sorte de vertébré. C'est l'ajout de nouvelles possibilités qui détermine les divisions de chaque branche.

Nous pouvons aussi souvent considérer un objet complexe comme la somme des capacités d'objets plus simples. Ainsi le biologiste contrairement au zoologue ne verra pas l'animal comme une spécialisation d'une entité plus générale mais comme la juxtaposition de fonctions : l'animal possède une fonction digestive, une fonction respiratoire, une fonction circulatoire,... De la même manière nous pouvons voir les objets comme la somme d'objets plus simples, chacun représentant une *fonction primaire* de l'objet.

[2]Les destructeurs étant toujours sans arguments, il n'y a pas de choix du destructeur comme il y a un choix du constructeur.

Souvent, de même qu'un appareil digestif n'existe pas isolé mais seulement dans un animal particulier, nos classes primaires n'auront pas d'objets autonomes, leurs objets seront des membres des classes dérivées.

De telles classes seront dites abstraites.

Un exemple de décomposition tirée de l'analyse d'une société effectuant du transport de marchandises en vrac est donné figure 8.5.

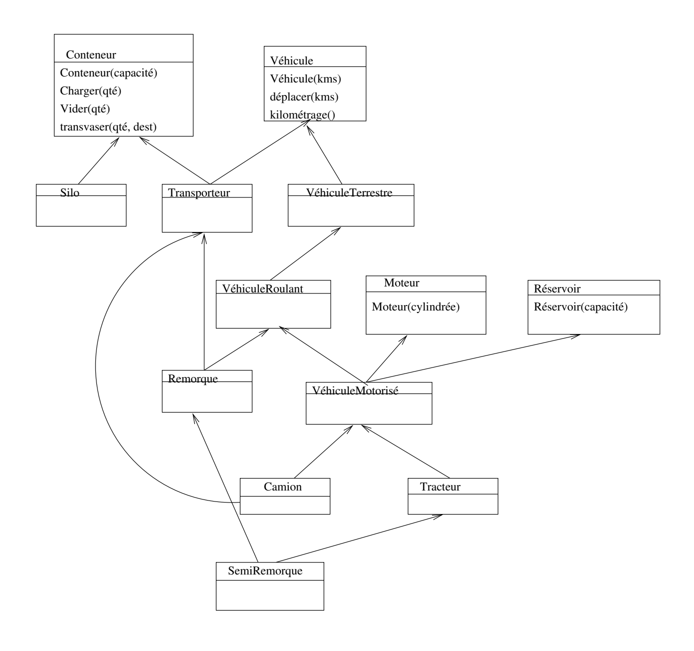


FIG. 8.5 – analyse d'une société de transport en vrac

Dans ce diagramme, les éléments de la classe `Transporteur`, qui sont les véhicules qui peuvent recevoir un chargement, ont à la fois les méthodes des `Véhicule`s et des `Conteneurs`.

La déclaration de la classe `Transporteur` peut être :

```
class Transporteur: public Conteneur, public Véhicule{
Transporteur( float capacité, float kilométrage=0.);
......
};
inline Transporteur::Transporteur( float capacité,
                                   float kilométrage=0.):
    Conteneur(capacité),
    Véhicule( kilométrage){
}
```

Nous pouvons alors pour un `Transporteur`, ou un membre d'une de ses classes dérivées, appeler une méthode qui appartient au protocole d'une de ses classes mères.

Par exemple :

```
Camion: c1(5000,  ... );
Silo s1(50000.);
Silo s2(100000.);
  ....
//charger le camion c1 par le contenu du silo s1
s1.transvaser(4000, c1); //méthode de Conteneur
s1.rouler(3000); //méthode de véhiculeRoulant
  ...
//décharger c1 dans le silo s2
c1.transvaser(4000, s2);
```

Nous obtenons une solution élégante et fiable car la gestion des stocks reste cantonnée à la classe silo. Sans l'héritage multiple nous aurions dû voir les classes `Transporteur` et `Silo` comme deux classes étrangères, chacune gérant un stock indépendant. L'opération `transvaser` aurait alors été une fonction amie de l'une ou des deux classes.

Mais *il est toujours préférable que l'état d'un objet ne puisse être modifié que par ses fonctions membres*. Cela permet de localiser les contrôles d'invariants à l'intérieur de la classe et simplifie ainsi le débogage.

8.3.2 Nom de méthode ambigu

Parfois deux classes de base peuvent offrir des méthodes qui ont le même nom, par exemple si nous définissons une méthode `afficher()` à la fois dans `Conteneur` et `Véhicule`, l'appel de `afficher()` dans la classe `Transporteur` ou une de ses descendantes est ambigu.

Nous définirons alors dans la classe `Transporteur` une méthode qui appelle explicitement chacune de ses fonctions par leur nom complet :

```
void Transporteur::afficher(){
   Conteneur::afficher();
   Véhicule::afficher();
}
```

8.3.3 Classe Virtuelle

Considérons la classe `Remorque` de la figure 8.5.

Si nous la déclarons sous la forme :

```
class Remorque: public VéhiculeRoulant,
                public Transporteur{ .... };
```

un objet de cette classe hérite de deux `Véhicule`s, l'un par le biais de `Transporteur`, l'autre par `VéhiculeRoulant` et `VéhiculeTerrestre`. Cela implique que toutes les données de `Véhicule` sont dupliquées dans `Remorque` et aussi que l'appel à une méthode telle `kilométrage()` est ambigu. De plus rien ne garantit que les états de ces deux véhicules de base soient identiques.

Puisque dans la réalité à une remorque correspond un seul véhicule il est possible de préciser par le mot clé `virtual` que la classe `Véhicule` ne fournit qu'un objet même si elle peut être atteinte par plusieurs chemins.

Les déclarations deviennent donc :

```
class VéhiculeTerrestre: public virtual Véhicule{ .... };
class VéhiculeRoulant: public  VéhiculeTerrestre{ .... };
class Transporteur: public Conteneur,
                    public virtual Véhicule{... };
class Remorque: public VéhiculeRoulant,
                public Transporteur{ .... };
```

Le cas de `SemiRemorque` est différent. L'analyse de l'application indique qu'il hérite de deux `VéhiculeRoulant`s distincts. Nous ne voulons pas confondre leurs états . Il n'y a aucune raison pour que la `Remorque` et le `Tracteur` aient le même kilométrage.

Aussi l'ambiguïté de la méthode `kilométrage()` et de la méthode `rouler(float)` devront être levées en indiquant explicitement à quel objet elles s'appliquent :

```
inline float SemiRemorque::kilométrage(){
  return Tracteur::kilométrage();
}

inline void SemiRemorque::rouler(float distance){
  Tracteur::rouler(distance);
  Remorque::rouler(distance);
}
```

Règle 10.5 Si l'on hérite du même parent par plusieurs classes de base, ce parent doit être une classe de base virtuelle.

Cette règle nous permet de déterminer quand nous devons penser à un héritage virtuel. Ici `Véhicule` est dérivé des deux classes de base `Transporteur` et `VéhiculeTerrestre` il doit donc être déclaré d'héritage virtuel.

Dans les cas complexes un examen de l'application est cependant nécessaire pour déterminer si un objet hérite d'un autre objet par deux chemins, ou s'il hérite de deux objets distincts de la même classe. C'est ce qu'illustre le cas `SemiRemorque`.

8.4 Méthode virtuelle

8.4.1 Les insuffisances du typage statique.

Si nous voulons garder trace de tous les éléments de la classe véhicule, nous les conserverons dans un conteneur, par exemple une liste.

Nous utiliserons un membre statique `listVehicM` d'un type conteneur que nous nommerons `ListMembres`.

Pour notre exemple nous choisissons comme conteneur le type patron `list` de la STL, et nous définissons donc :

```
typedef list<const Véhicule*> ListMembres;
```

Il est alors possible pour une méthode statique de va classe `Véhicule` d'accéder à tous les véhicules.

Nous prendrons pour exemple une fonction `toutAfficher` qui affiche tous les véhicules définis dans la classe.

```
class Véhicule{
public:
  // ajoute le véhicule à la liste
  Véhicule(float kmInit);
  // supprime le  véhicule de la liste
  ~Véhicule();
  // affiche le véhicule courant
  void afficher() const;
  // affiche tous les véhicules
  static void toutAfficher();
private:
  float distanceM;
  typedef list<const Véhicule*>
                     ListMembres;
  // Liste de tous les véhicules
  static  ListMembres listVehicM;
};
```

Pour tenir à jour la liste, les constructeurs et destructeurs doivent être modifiés de manière à insérer et supprimer l'objet courant de la liste de tous les objets.

```
Véhicule::ListMembres Véhicule::listVehicM; //initialisation

Véhicule::Véhicule(float kmInit):
    distanceM( kmInit)
{
    listVehicM.push_front(this);
}
Véhicule::~Véhicule(){
  ListMembres::iterator it=
    find(listVehicM.begin(),
         listVehicM.end(),this);
  listVehicM.erase(it);
}
```

La fonction `toutAfficher()`, quant à elle, est un simple parcours de liste qui appelle `afficher()` sur chaque objet.

```
void  Véhicule::toutAfficher(){
  for( ListMembres::const_iterator
          it(listVehicM.begin());
       it!=listVehicM.end();++it){
     (*it)->afficher();
  }
}
```

ou encore si on utilise `<algorithm>` de la STL :[3]

```
for_each(listVehicM.begin(),
         listVehicM.end(),
         mem_fun(&Vehicule::afficher));
```

Mais ici l'appel de la fonction `afficher()` va se révéler très décevant : puisque nous utilisons un pointeur de `Véhicule` pour parcourir la liste, ce sera toujours la fonction `afficher()` de la classe `Véhicule` qui sera appelée.

Le compilateur ne peut pas connaître la sous classe qui est désignée puisque celle-ci n'est pas déterminée à la compilation ; le type de l'objet pointé par `vP` n'est connu que dynamiquement au moment de l'execution. Le compilateur utilise les informations de type qu'il possède et génère un appel à la fonction `Véhicule::afficher()`. L' affichage sera donc limité à la distance qui est la seule caractéristique commune à tous les véhicules.

[3]Voir §46 pour l'adaptateur `mem_fun` et §10.2.1 pour l'algorithme `for_each`

8.4.2 Typage dynamique

Pour résoudre le type de problème évoqué dans le paragraphe précédant, il nous faut de retarder la liaison des fonctions jusqu'au moment de l'exécution ; à ce moment il est possible de déterminer dynamiquement quelle est la classe de l'objet pointé, et d'appeler la fonction correspondante.

Le mot-clé `virtual` de **C++** nous permet d'indiquer au compilateur que la fonction désignée doit avoir une liaison dynamique et non statique.

Nous écrirons donc :

```
class Véhicule{
  virtual void afficher();
};
class Transporteur: public Conteneur, public Véhicule{
  virtual void afficher();
};
class Camion: public Transporteur{
  virtual void afficher();
};
```

Quand le pointeur :

```
Véhicule * vP;
```

pointe sur un camion, l'appel de :

```
  vP->afficher();
```

va maintenant appeler la fonction `afficher` de la classe camion et écrire :

```
  Camion immatriculé 1234 RR 22
  Charge utile : 15 T
.....
```

De telles fonctions sont dites *virtuelles* car au moment de la compilation seule leur déclaration est connue. La définition qui sera employée n'est connue qu'à l'exécution.

Notons que si nous avons comme définition :

```
void Transporteur::afficher(){
   Conteneur::afficher();
   Véhicule::afficher();
}
```

les appels `Conteneur::afficher()` et `Véhicule::afficher()` *n'utilisent pas le mécanisme virtuel*. La qualification de la fonction par le nom de la classe désactive le mécanisme virtuel et évite ainsi de boucler.

8.4.3 Appel calculé et liaison dynamique

Rec. 10.3 Les instructions de sélection `if-else` et `switch` seront utilisées quand le flux de contrôle dépend de la valeur de l'objet ; la liaison dynamique est utilisée quand le flux de contrôle dépend du type de l'objet.

Dans le cas ou le contrôle dépend du type de l'objet, utiliser un drapeau pour déterminer ce type est une solution à rejeter quand le langage admet la liaison dynamique.

En effet si on modifie ou on étend l'arbre d'héritage , il faut alors modifier aussi les instructions de sélection de la classe de base. Cela n'est pas toujours possible si nous ne possédons pas le code source de ces classes et cela limite la modularité du développement.

8.4.4 Comparaison des fonctions à lien statique et à lien dynamique

– L'appel des méthodes virtuelles est un peu plus coûteux que celui d'une méthode simple, car il se fait par l'intermédiaire d'une table des fonctions virtuelles qui est associée à toute classe comprenant des méthodes virtuelles.
– Les objets d'une classe à méthode virtuelle comportent donc en plus des données, un pointeur sur la table des méthodes virtuelles.
– Les fonctions virtuelles compliquent le débogage, puisqu'on ne peut plus déterminer à la lecture du texte source quelle fonction va être appelée.
– Les fonctions virtuelles ne peuvent pas être développées en-ligne, puisque la fonction utilisée est inconnue du compilateur.
– Cependant quand l'opérateur `::` désactive le mode virtuel le compilateur retrouve la possibilité de remplacer en ligne les fonctions (même virtuelles) déclarées `inline`.
  Ce n'est donc pas une erreur de déclarer `inline` une fonction virtuelle, mais cette déclaration est ignorée pour les appels qui sont à lien dynamique.

8.4.5 Redéfinition d'une méthode virtuelle

Une méthode virtuelle est redéfinie dans les classes filles par les méthodes qui ont le même nom, les mêmes nombre et types d'arguments.

Mais pour que cette redéfinition soit correcte il faut que la méthode de la classe fille rende une valeur de même type que la fonction d'origine, ou bien un pointeur ou référence sur une classe dérivée du type de retour initial.
Exemple :

```
class A{
  public:
    virtual int f(int);
    virtual float g (float);
    virtual A* h (A*);
    virtual A& k();
};
class B: public A{
  public:
    virtual int f(int);     // redéfini A::f
    virtual void g(float);  // erreur g cache A::g(float)
                            // sans le redéfinir
    virtual B* h (B*);      // h ne cache pas A::h(A*)
                            //  et ne le redéfini pas non plus.
    virtual B& k();         // redéfini A::k();
};
```

8.4.6 Destructeurs virtuels

Dans une classe comprenant des fonctions virtuelles, le destructeur devrait être déclaré virtuel ; cela permet à un `delete` d'appliquer le destructeur approprié à l'objet pointé même quand on accède à celui-ci par un pointeur sur une classe de base.

Règle 10.4 Une classe de base publique doit avoir un destructeur virtuel ou un destructeur protégé.

Par exemple si nous avons créé un véhicule par :

```
Véhicule* vP = new Camion( ...  );
```

et que nous le détruisons par :

```
delete vP;
```

le destructeur de `Véhicule` est appelé sur l'objet désigné par `vP`. Si ce destructeur est une fonction à liaison statique elle ne pourra supprimer les attributs spécifiques d'un `Camion`. En revanche un destructeur à liaison dynamique (*i.e.* déclaré `virtual`) pourra détruire les membres de la classe de l'objet et successivement tous les membres des classes de base en remontant dans l'arbre d'héritage.

Certaines classes par contre ne sont utilisées que pour factoriser une partie du protocole d'autres classes et n'ont pas d'instances (Elles sont à distinguer des

classes abstraites qui ont des instances dans les classes dérivées mais dont on ne connaît pas le type à la compilation).

Ces classe n'ont pas d'objets et on n'accède jamais à un objet par un pointeur de ces classes. Elles ont leur constructeurs et leurs destructeurs protégés (et non virtuels !) puisqu'elles ne sont créées et détruites que que par l'intermédiaire des classes filles.

8.4.7 Construction par clonage

Puisqu'une définition d'objet comprend toujours le type de l'objet créé, il n'y a pas de constructeur virtuel.

Cependant il est fréquent de devoir dupliquer un objet sur lequel nous possédons un pointeur, tel l'objet pointé par `pv` ci-dessous

```
Véhicule *pv;
...
pv=new Camion( 50000, "1234RR22", 10000);
```

L'emploi d' un constructeur de copie n'est pas satisfaisant si nous ne connaissons pas le type de l'objet à dupliquer.

La ligne :

```
 Véhicule copieVéhicule(*pv);
```

ne va copier que la partie de `pv` qui figure dans véhicule.

Nous pouvons définir une opération virtuelle de *clonage* qui duplique l'objet courant et rend un pointeur sur une copie placée en mémoire libre.

```
class Véhicule{
  public:
    virtual Véhicule* cloner()const{
      return new Véhicule(*this);
    }
  ....
};

class Transporteur: public Conteneur, public Véhicule{
  public:
    virtual Transporteur* cloner()const{
      return new Transporteur(*this);
    }
   .....
```

```
};


class Camion:  public Transporteur{
  public:
    virtual Camion* clonerVéhicule()const{
       return new Camion(*this);
    }
  ......
};
```

Avec ces définitions [4] nous sommes assurés que nous obtiendrons bien une copie du véhicule concerné avec :

```
Véhicule* vPcopie = vP->cloner();
```

8.4.8 fonction polymorphe

Une fonction qui admet une référence ou un pointeur sur une classe possédant une méthode virtuelle est dite *polymorphe*, en effet le type réel de son argument ne peut être déterminé qu'a l'exécution.

Une fonction polymorphe a donc un typage dynamique et non statique comme les fonctions « standards » **C**++ .

Le polymorphisme ne doit pas être confondu avec la surcharge dans laquelle plusieurs fonctions partagent le même nom. Pour la surcharge le type des arguments et la fonction appelée sont connus à la compilation, le typage est donc statique.

8.4.9 Classe abstraite

Une classe abstraite est une classe qui n'a pas d'objets propres, c'est à dire dont tous les objets sont membres d'une classe dérivée.

Souvent dans un diagramme d'héritage les classes situées au sommet du diagramme sont des classes abstraites qui servent à regrouper des objets qui possèdent un protocole commun.

Comme nous souhaitons connaître le type exact de tout véhicule ; chaque véhicule sera aussi une voiture, un camion, un bateau à voile ou une trottinette.

Il n'y a aucun véhicule qui ne soit *que* `Véhicule` La classe `Véhicule` est donc *abstraite*.

---

[4] `Transporteur : :cloner()` redéfinit à la fois `Véhicule : :cloner()` et un éventuel `Conteneur : :cloner()`

Les classes abstraites n'ont pas de constructeur public. Les objets sont construits dans les constructeurs protégés que les classes dérivées appellent.

8.4.10 Méthode purement virtuelle

Certaines méthodes déclarées dans une classe abstraite ne peuvent être *définies* car leur algorithme met en jeu les caractéristiques des sous classes. Cependant on peut assurer que tous les objets des classes dérivées devront fournir la fonction, elle fait donc partie du protocole de la classe.

On marquera que ces fonctions virtuelles sont déclarées mais ne sont définies que dans les sous-classes en leur affectant la valeur `0` dans leur déclaration.

Une telle fonction est dite *purement virtuelle*.

Exemple :

```
class Véhicule{
public:
  void afficher()=0; //pas assez d'information pour afficher
}
```

Chapitre 9

Patrons et conteneurs

9.1 Conteneurs et itérateurs

Un conteneur est un objet qui *contient* d'autres objets et offre un accès ordonné aux objets contenus. Cet accès est fourni par un *itérateur* qui est une sorte d'index qui se déplace à l'intérieur du conteneur.[1] Il y a différentes manières de déplacer l'itérateur, vers l'avant, vers l'arrière, en un point quelconque, etc. Il y a aussi différentes possibilités quant aux places ou il est possible d'insérer ou de supprimer un objet.

Un conteneur doit choisir les opérations dont il veut privilégier l'efficacité, et adapter son modèle de gestion mémoire en fonction de ses priorités. Différentes sortes de conteneurs offrent ainsi différents jeux de primitives pour déplacer les itérateurs, ainsi que différentes possibilités d'insertions et de suppressions d'objets.

La structure même de l'objet stocké, sa classe, son protocole, n'ont pas besoin d'être connu par le gestionnaire mémoire sous-jacent au conteneur ; pour lui un objet est surtout l'occupant d'une place mémoire, le gestionnaire mémoire loue un *box* il n'a pas besoin de savoir quelle sorte de véhicule vous y garez seules ses dimensions lui importent. D'un autre coté les primitives d'accès aux objets doivent rendre un objet typé, de manière à garantir un emploi cohérent de cet objet.

Nous avons donc deux contraintes contradictoires, nous voulons ignorer le type de l'objet (à l'exception peut-être de sa taille) pour pouvoir concevoir des conteneurs à usages multiples, mais nous voulons préserver le mécanisme de typage.

[1]Certains conteneurs peuvent ne permettre d'accéder qu'à un seul objet, les autres restant *cachés* jusqu'à ce qu'un changement d'état du conteneur les dévoile. Ainsi une pile peut ne laisser voir que son premier objet.

À ce problème les différents langages répondent par deux sortes de solutions.

Les conteneurs dans les langages à typage dynamique.

Les langages à typage dynamique tels *Smalltalk*, *CLOS*, mais aussi *Java* fournissent des conteneurs du type passe-partout *Objet*, et chaque objet du conteneur auquel on accède déclare dynamiquement son type.

Cette solution est aussi utilisable en **C**++ si on choisi aussi d'adopter un typage dynamique et de placer toutes ses classes sous la même classe racine.

Cette solution offre une grande flexibilité et permet même de traiter aisément les conteneurs hétérogènes.

En revanche elle oblige à ne stocker dans la représentation mémoire du conteneur qu'une référence à un objet dont la taille n'est pas connue par avance, et les objets doivent porter un pointeur vers leur type comme toujours dans le typage dynamique. Elle est donc frappée d'une double inefficacité pour les petits objets, inefficacité d'accès à cause de l'indirection et inefficacité de stockage à cause du pointeur ajouté.

Mais le principal désavantage de cette solution est qu'elle retarde tout les contrôles de type jusqu'à l'exécution rendant ainsi impossible une validation par le compilateur.

Les conteneurs dans les langages à typage statique.

Si nous voulons, ou nous devons garder un typage statique l'information sur le type d'objet contenu doit être partie intégrante du conteneur.

Il est alors possible de concevoir un conteneur distinct pour chaque type. On produit dans ce cas des classes similaires mais indépendantes dont les fonctions membres sont dupliquées pour chaque type. La taille du code augmente mais souvent la taille du code n'est pas critique sur les machines modernes. Cependant la multiplication des classes indépendantes produites augmente le travail de programmation, la complexité du programme et la fragilité de celui-ci.

Pour pallier les déficiences de ce type d'approche les programmeurs ont essayé de regrouper ces classes et méthodes apparentées.

Une solution est de concevoir une classe conteneur non typée, et de l'*habiller* à la demande avec un nouveau typage pour chaque classe de contenu requis. Cette méthode réduit le volume du code et permet de faire la conception de chaque classe conteneur une seule fois. Mais, malgré leur similarité, les *classes interfaces* qui servent de façade doivent toujours être reproduites pour chaque type de contenu.

Pour générer celles-ci on à tout d'abord utilisé les macro-instructions, procédé dont les inconvénients sont rappelés dans le paragraphe suivant §9.2.

Le mécanisme des patrons permet de générer les conteneurs de la manière la plus économique et surtout la plus sure. et il voit dans la conception de ceux-ci sa principale utilité, c'est pourquoi nous traiterons de concert conteneurs et patrons, d'autant plus que la bibliothèque standard des patrons *Standard Template Library* est aussi celle qui fournit les conteneurs standards.

9.2 Utilité des Patrons

Nous allons dans cette section traiter du mécanisme de reproduction des *patrons*.

Le mécanisme des fonctions permet d'effectuer un calcul qui dépend d'un paramètre qui varie parmi les différentes valeurs possibles d'un type donné.

Mais fréquemment nous aurons besoin d'un niveau d'abstraction supplémentaire, les fonctions semblables différeront, non par la valeur d'un paramètre, mais par le *type* même du paramètre. Dans le cas le plus commun le type peut être déterminé lors de la compilation pour chaque appel de fonction. Le compilateur peut alors effectuer les vérifications de cohérence des types qui caractérisent les langages fortement typés. [2]

Une réponse simple et ancienne à ce problème est l'emploi de macro-instructions (cf. §4.3.1) dans lesquelles le type employé est un argument textuel qui est instantié lors de la pré-compilation.

Ce système présente cependant de nombreux désavantages provenant du remplacement aveugle des paramètres patrons *avant* la compilation, et donc à un moment où aucune information sur la sémantique des objets n'est disponible et alors que même la catégorie lexicale n'est pas encore déterminée. L'expansion des macro-instructions n'est en effet qu'un remplacement brutal de chaînes de caractères.

Les erreurs qui sont ensuite décelées à la compilation viennent du texte expansé, le seul vu par le compilateur. Ainsi le programmeur et le compilateur ont deux vues différentes du programme, nous voyons le texte avant expansion , en revanche le texte expansé est traité. En conséquence certaines erreurs ne sont pas décelées et la véritable cause des erreurs trouvées reste souvent obscure.

Le recours à une phase antérieure à la compilation a en outre l'inconvénient d'avoir la primeur sur les opérations effectuées par le compilateur, et de ne laisser à celui-ci aucune possibilité d'utiliser un appel de fonction qui viendrait en concurrence avec la macro-instruction, soit directement, soit par surcharge ou encore par conversion d'arguments.

[2] Dans le cas ou le type varie de manière dynamique d'un appel à l'autre on utilisera des fonctions polymorphes (voir §8.4.8).

9.3 Fonctions patrons

Le mécanisme des fonctions patrons vient pallier ce défaut des macro-instructions : il est pris en charge directement par le compilateur qui peut le réserver au cas où un appel *normal* ne peut être généré.

Nous illustrerons ce mécanisme par un exemple :

Supposons que nous ayons à copier des entiers d'un tableau dans un autre, nous pourrons utiliser :

```
void copier ( const int* deb1, const int* fin1,
              int* deb2, int* fin2)
{
  while (deb1 != fin1 && deb2 != fin2) {
     *deb2=*deb1;
     ++deb1;
     ++deb2;
}
```

Si nous voulons copier maintenant des caractères d'une zone mémoire dans une autre nous emploierons :

```
void copier ( const char* deb1, const char* fin1,
              char* deb2, char* fin2)
{
  while (deb1 != fin1 && deb2 != fin2) {
     *deb2=*deb1;
     ++deb1;
     ++deb2;
}
```

Si nous voulons employer une liste chaînée d'objets d'une classe quelconque `Data` nous inclurons ces `Data` dans une cellule qui possède en outre un pointeur sur la cellule suivante :

```
class Cell{
public:
    Data content;
private:
    Cell* suiv; //cellule suivante dans une liste
}
```

La liste d'objets de la classe `Cell` est repérée par les pointeurs `deb1` et `fin1` (`fin1` pointe **après** le dernier élément de la liste). Ses éléments seront recopiés en remplacement de ceux de la liste repérée par `deb2` et `fin2` par : [3]

```
void copier ( const Cell* deb1, const Cell* fin1, Cell* deb2){
  while (deb1!=fin1&& deb2!=fin2) {
     *deb2.content=*deb1.content;
     deb1=deb1.suiv;
     deb2=deb2.suiv;
}
```

Pour éviter de de surcharger de nombreuses versions de la fonction *copier* nous utiliserons le mécanisme des fonctions patrons. Il nous permet d'écrire :

```
template <class InputIterator, class OutputIterator>
inline void copier ( InputIterator deb1, InputIterator fin1,
                     OutputIterator deb2, OutputIterator fin2){
  while (deb1!=fin1 && deb2 != fin2) {
     *deb2=*deb1;
     ++deb1;
     ++deb2;
  }
}
```

Pour utiliser cette fonction, les objets de type `InputIterator` et `Output-Iterator` doivent pouvoir être comparés avec l'opérateur `!=` ; les objets des types `InputIterator` et `OutputIterator` doivent pouvoir être *déréférencés* par l'opérateur `*`, l'opérateur préfixe `++` doit permettre de passer à l'élément suivant, et enfin une affectation de l'objet référencé par un `InputIterator` vers celui référencé par un `OutputIterator` doit être possible et avoir la signification attendue pour une copie des données.

Ces conditions sont réalisées par des pointeurs sur des zones allouées en mémoire quand la classe `Data` a une affectation adéquate , mais nous pouvons définir des itérateurs qui peuvent être traités par cet algorithme pour de nombreux autres types de *conteneurs*. Nous verrons ci-dessous en §9.3.1 l'exemple des listes chaînées.

[3] Pour que la copie soit possible il est nécessaire que l'opérateur d'affectation de la classe `Data` effectue la copie d'une la manière correcte, il convient donc dans ce cas de vérifier que le type de copie souhaité (superficiel ou en profondeur, voir 7.6.2 et 7.6.3) est effectivement utilisé.

9.3.1 Argument Patron

On nomme **argument patron** un argument formel placé entre crochets `<` et `>` dans une définition de fonction patron. Les arguments patrons peuvent être des noms de types , des adresses d'objets statiques ou encore des expressions constantes.

Le spécificateur `class` devant un argument formel indique un

type, et contrairement à ce que semble impliquer le mot-clé ce n'est pas obligatoirement une classe, ce On utilisera donc la fonction `copier` de l'exemple précédent avec :

```
const char s[]="toto";
char t[40];
copier(s, s+4, t, t+39 );
```

dans lequel l'argument patron formel `InputIterator` est remplacé `const char*` et l'argument patron formel `OutputIterator` est remplacé `char*`.

Les Pointeurs ne sont pas adaptés au Type `Cell` précédant, pour lequel nous avons besoin d'un itérateur qui *pointe* sur les données et qui *avance* d'une cellule à la suivante.

Nous définissons donc le type :

```
Class InputListIterator{
   public:
      InputListIterator();
      InputListIterator(const InputListIterator& li);
      InputListIterator(Cell* pcell);
      InputListIterator operator ++ ();
      bool operator != (InputListIterator it);
      const Data& operator *()const;
   protected:
      Cell* cellpt;
};
```

Un exemple d'utilisation de la fonction *copier* est :

```
InputListIterator deb1, fin1;
OutputListIterator deb2,fin2;
......   //création des listes
copier(deb1,fin1,deb2,fin2);
```

La fonction de déréférence de `InputIterator` s'écrira comme celle des pointeurs sur des constantes :

```
const Data& InputListIterator::operator *()const{
  return cellpt->content;
}
```

en revanche la fonction d'incrémentation se distingue de celle des pointeurs et devient :

```
InputListIterator InputListIterator::operator ++ (){
    cellpt=cellpt->suiv.cellpt;
   return *this;
}
```

La classe `OutputListIterator` est dérivée de la classe `InputListIterator` mais l'opérateur `*` rend une référence *non constante* à la donnée.

```
Data& OutputListIterator::operator *()const{
  return cellpt->content;
}
```

9.4 Propriétés des types patrons

Pour qu'un patron puisse être généré. Les arguments *patrons* effectifs doivent pouvoir figurer à la place des arguments formels dans tous les appels de fonctions. Cette propriété des arguments patrons effectifs est vérifiée par le compilateur à chaque instantiation d'un exemplaire d'une fonction patron. Cette vérification de la cohérence des arguments patrons *à la compilation* se démarque de l'utilisation de macro-instructions pour lesquelles le remplacement se fait sans tenir compte du type des paramètres.

Par exemple dans la fonction patron :

```
template <class T> int f(T t){ return t.g();}
```

`T` doit être remplacé par une classe possédant une méthode `g()` qui rend un entier ou une valeur qui se convertit en entier. Ainsi à la lecture du fragment de programme :

```
A a;
int i;
i= f(a);
```

le compilateur va tenter de générer l'exemplaire :

```
int f(A);
```

et si la fonction `A::g(int)` n'existe pas, ou rend un type incorrect le compilateur générera un message d'erreur.

Les patrons permettent de vérifier à la compilation que l'argument effectif possède les méthodes nécessaires à la réalisation de la fonction patron.

Pour le concepteur un, argument patron est un *type abstrait* c'est-à-dire une classe munie d'un certain protocole, comprenant non seulement la *signature* des méthodes mais aussi leur spécification. Dans la documentation du patron les spécifications de ses arguments doivent accompagner la déclaration. Ces spécifications comprennent non seulement une signature [4] mais aussi la spécification logique de chaque opération avec ses pré et post-conditions. La présence des méthodes d'une signature appropriée est vérifiée par le compilateur **C**++ (mais pas les spécifications !).

Dans notre exemple précédent nous avons vu ci-dessus les contraintes imposées aux types `InputIterator` et `OutputIterator`. Le compilateur rejettera ainsi une tentative d'utiliser :

```
const int t[]={1,2,3,4,5};
const int u[]={11,12,13,14,15};
copier(t,T+5,u);
```

car l'affectation n'est pas autorisée.

Pour les listes constituées de `Cell` les pointeurs `Cell*` sont acceptés comme `OutputIterator` par le compilateur, mais ne produisent pas l'effet souhaité car l'opérateur ++ ne passe pas à la cellule suivante.

Les `InputIterator` et `OutputIterator` donnent à l'opérateur ++ le comportement souhaité.

Notons que les itérateurs qui marquent la *fin* des zones à copier doivent être situés *après* cette zone, cela ne pose pas de problème pour un tableau où l'on donnera l'adresse après la fin du tableau ; pour une liste, en revanche, un élément supplémentaire (dit *drapeau* ou *sentinelle*) doit être explicitement prévu en fin de liste.

Nous pouvons rendre la manipulation de ces listes plus aisée et plus fiable en fournissant une classe qui permet de les gérer. Une réalisation minimale serait :

```
class DataList{
  public:
   DataList(): beginM(new Cell()),endM(beginM){}
   OutputListIterator begin(){return beginM;}
   OutputListIterator end(){return endM;}
   void pushFront(const Data&);
```

[4] Le nom et le type des arguments de chaque méthode (voir §7.1).

```
  private:
   OutputListIterator beginM;
   OutputListIterator endM;
};
```

Une telle définition ne donne pas le moyen d'utiliser des `DataList` *constantes*, pour de telles listes, nous ne souhaitons ni permettre l'ajout d'éléments, ni la modification d'éléments ; nous ajouterons donc les méthodes :

```
   InputListIterator begin()const{return beginM;}
   InputListIterator end()const{return endM;}
```

Bien sûr nous pouvons enrichir cette classe pour fournir toutes les opérations d'ajout, de suppression, de recherche, de modification de liste habituellement utilisées ainsi que les extractions de sous-listes.

9.5 Génération des patrons

Nous avons vu que lors d'un appel de fonction le compilateur sélectionnait la fonction la plus appropriée grâce au mécanisme de surcharge des fonctions, ce qui constitue l'aspect statique de la sélection des fonctions.

Le mécanisme du polymorphisme permet aussi de manière dynamique de passer un paramètre à une fonction déclarée avec un type qui est un antécédent du type donné dans l'arbre d'héritage.

Les fonctions patrons qui sont générées automatiquement par le compilateur viennent apporter une troisième possibilité de résoudre un appel de fonction.

9.5.1 Mécanisme de résolution des patrons.

Le compilateur génère une fonction patron quand il rencontre un appel qui ne peut pas être résolu avec une fonction déclarée de même nom et qui comporte exactement les types des arguments ; mais qu'il peut trouver une combinaison d'arguments telle que la fonction patron générée puisse être appliquée **sans conversion d'arguments.**

Par exemple pour :

```
const char s1[]="exemple";
char s2[30];
copier ( s1,s1+strlen(s1), s2);
```

le compilateur générera la fonction :

```
void copier ( const char* deb1, const char* fin1, char* deb2);
```

en instantiant le paramètre patron `InputIterator` à `const char*` et le paramètre patron `OutputIterator` à la `char*`.

Il est important de se souvenir dans l'utilisation des patrons de la stratégie d'instantiation du compilateur :

1. Essayer les fonctions déclarées **sans conversion** d'arguments.
2. Générer les exemplaires de fonctions patron **sans conversion** d'arguments.

   Si plusieurs exemplaires conviennent le compilateur choisi toujours *le plus spécialisé*, si en revanche il y a le choix entre plusieurs exemplaires et qu'aucun n'est plus spécialisé que l'autre, c'est un cas d'ambiguïté qui provoque une erreur de compilation.
3. Essayer les fonctions déclarées **avec conversion** d'arguments

La génération d'une fonction patron se fait donc toujours sans tenir compte des conversions.

Par exemple la fonction patron :

```
template <class Iterator>
inline void copier ( const Iterator deb1,
                     const Iterator fin1, Iterator deb2){
  while (deb1!=fin1) *deb2++=*deb1++;
}
```

permet de résoudre l'exemple précédent, mais ne permet pas de traiter :

```
char s1[]="exemple";
char s2[30];
copier ( s1,s1+strlen(s1), s2);
```

qui demande la fonction :

```
void copier (char*, char*, char*);
```

qui ne peut pas être générée.

Bien que la fonction :

```
void copier (const char*, const char*, char*);
```

convienne, le compilateur ne la générera pas car elle demande une conversion.

Programme 9.1 Classe patron pour les paires d'objets

```
namespace mzlib{
template <typename T, class U> class Cons{
   public:
      typedef Cons<T1,T2> Self;
      typedef T1 Car;
      typedef T2 Cdr;
      static const int length=Cdr::length+1;
      static const bool null=false;
      Cons(const T1& t1, const T2& t2):carM(t1),cdrM(t2){}
      //default ctor
      Cons():carM(),cdrM(){}
      const Car& car()const{return carM;}
      Car& car(){return carM;}
      const Cdr& cdr()const{return cdrM;}
      Cdr& cdr(){return cdrM;}
   private:
      Car carM;
      Cdr cdrM;
};

}//end namespace mzlib
```

9.6 Classes patrons.

De la même manière que nous pouvons utiliser des modèles de fonctions qui suivant le type des arguments pourront être spécialisées en une famille de fonction, nous pouvons aussi avoir des modèles de classe qui dépendront de la valeur des types d'*arguments patrons*.

Comme nous l'avons vu précédemment (§**??**) les classes conteneurs, pour être paramétrées par le type d'objet contenu, doivent être déclarées comme classe patron.

Nous prendrons comme exemple de classe patron la classe `Cons` du programme 9.1. Cette classe est proche de la classe `pair` de la STL, mais pour éviter les confusions tous les noms sont différents.

Pour construire un `Cons` on doit donner des arguments patrons de manière à instancier la spécialisation correcte de la classe patron. On écrira par exemple :

```
Cons<int, double> unePaire(1, 3.14);
```

Nous pourrons tirer parti de la capacité du compilateur de déduire les types patrons d'une *fonction* patron à partir du type de ses arguments en créant une fonction

Programme 9.2 `Cons` : fonction patron auxiliaire

```
namespace mzlib{
template <class T1, class T2> inline
Cons<T1, T2> cons(const T1& t1, const T2& t2){
   return Cons<T1, T2>(t1,t2);
}
} //end namesdpace mzlib


int main(){
   using namespace mzlib;
   using std::cout; using std::endl;
   int i=1;
   double x=3.14159653;
   cout<< cons(i,make_list(x)).car()<<", "cons(i,make_list(x)).cdr()<<endl;
}
```

Résultat

```
1, 3.1416
```

patron qui initialise un nouvel objet (Programme 9.2).

Cette fonction patron, nommée ici `cons`, ne fait rien d'autre que d'appeler le constructeur : elle permet de ne pas avoir à donner explicitement les arguments patrons. Ainsi `cons(1,3.14)` est identique à `Cons<int,double>(1,3.14)`.

Ces fonctions patrons de construction rendent l'utilisation des classes patrons plus aisée. Elles sont utilisées dans la bibliothèque standard, en particulier pour les `pair`es (§10.3.2), et pour les différents adaptateurs (§10.4.3).

Nous pouvons utiliser la classe `Cons` pour apparier différents types d'objets, en particulier nous pouvons nous intéresser aux `Cons` dont le deuxième membre

Programme 9.3 Structure Nil

```
namespace mzlib{
struct Nil{
      static const int length=0;
      static const bool null=true;
      std::ostream& put(std::ostream& os){
         return os;
      }
};
```

Programme 9.4 `Cons` spécialisation partielle.

```
namespace mzlib{
template <typename T> class Cons<T,Nil>{
   public:
      typedef Cons<T,Nil> Self;
      typedef T Car;
      typedef Nil Cdr;
      static const int length=Cdr::length+1;
      static const bool null=false;
      Cons(const T& t, const Nil& u):carM(t){}
      //default ctor
      Cons():carM(){}
      const Car& car()const{return carM;}
      Car& car(){return carM;}
      Cdr cdr()const{return Nil();}
      std::ostream& put(std::ostream& os){
         os<<car()<<", ";
         cdr().put(os);
      }
   private:
      Car carM;
};
} //end namespace mzlib
```

est lui-même un `Cons`. Nous pouvons ainsi représenter des listes d'objets : c'est l'idée originale à partir de laquelle à été bâti le langage LISP dans les années 60[5].

Puisqu'il faut que nos listes se terminent le dernier élément ne peut être un `Cons`, il faut que cela soit un type qui représente une liste vide, par exemple la structure `Nil` du programme 9.3.

Les listes à un seul élément sont représentées par un `Cons` dont le deuxième membre est `Nil`, et si nous regardons la classe `Cons` nous voyons que dans ce cas le second membre `cdrM` contient une instance de l'objet `Nil()`. C'est là encore ainsi que procèdent les LISP ou SCHEME. Cela explique que si un entier prend un mot mémoire, une liste qui ne contient qu'un entier prend deux mots mémoire.

Tout cela est normal dans les langages non typés interprétés, mais **C**$^{++}$ est compilé et fortement typé. Quand le compilateur voit une liste à un élément *il sait* qu'il s'agit d'un élément de type `Cons(T,Nil)`, il n'a pas besoin de consulter la mémoire pour cela. Autrement dit pour ce cas si fréquent des listes à un élément,

[5]Le nom *cons* est une abréviation de *construct* à été employé dès le début de LISP, comme les mots *car* et *cdr* qui désignaient les deux demi-mots de l' IBM 704 alors utilisé.

Programme 9.5 Utilisation de la classe `Cons`.

```
#include <iostream>
using std::cout; using std::endl;
using namespace mzlib;
template <class A1, class A2, class A3> inline
Cons<A1,Cons<A2,Cons<A3,Nil()> > >::value_type
   make_list( A1 a1,A2 a2, A3 a3)
{
   return cons(a1,make_list(a2,a3));
}
int main(){
    cout<< make_list(4,5.55,6)<<": size=";
    cout<<sizeof(make_list(4,5.55,6))/sizeof(int)<<endl;
```

Résultat

```
4, 5.55, 6, : size=4
```

le champ `cdrM` est inutile. Il est possible de *spécialiser* notre patron, en donnant une version particulière de la classe dans le cas ou le second type est `Nil`. Comme le premier argument peut toujours varier il s'agit d'une spécialisation partielle. Cette spécialisation est donnée dans le programme 9.4.

Un exemple d'utilisation des listes de `Cons` est donné dans le programme 9.5, qui montre qu'une liste comprenant deux entiers et un double, prend 4 mots en mémoire, ce qui correspond exactement à la place prise par ses trois éléments, sans aucun surcoût.

Nos listes, ne correspondent pas à une classe unique, comme cela serait sans doute le cas si nous utilisions le typage dynamique. Tous les objets considérés comme des listes ne sont même pas membre de la même famille patron, ce qui distingue nos listes c'est le *protocole* des objets liste. Ce protocole notre conception doit le fixer, dans notre cas nous déciderons d'appeler liste un objet qui possède les opérations suivantes :

- un booléen `null` vrai si c'est la liste triviale vide,
- un entier positif `length` qui donne sa longueur, c'est a dire 0 quand la liste est `null` et $1+cdr().length$ sinon.
- si `null` est faux : deux méthodes `car()` et `cdr()` telles que `cdr()` soit aussi une liste.

Ce protocole peut être vérifié *sur le code* même des classes qui prétendent fournir des listes.

Bien sûr si nous avons une classe patron qui attend une classe *liste* en argument patron, le fait d'écrire : `template <class Liste>` ne peut garantir que

Programme 9.6 Cons : adresse du n-ième élément.

```
template < int n, class L> struct GetNth{
      typedef GetNth<n-1,typename L::Cdr> GetNext;
      typedef typename  GetNext::value_type value_type;
      value_type operator()(const L& l){
         return GetNext()(l.cdr());
      }
};
template <>
template < class L> struct GetNth<1,L>{
      typedef typename L::Car value_type;
      value_type  operator()(const L& l){
         return l.car();
      }
};
template < int n, class L>
typename GetNth<n,L>::value_type getnth(const L& l){
   return GetNth<n,L>()(l);
}
```

Liste conviendra. Lors de la génération de l'exemplaire une erreur de compilation peut se produire parce que des méthodes manquent. À l'exécution aussi, des spécifications non respectées peuvent provoquer des erreurs.

Le compilateur commence par instantiatier les paramètres patrons et générer l'exemplaire ; pour détecter dès le début les erreurs qui peuvent être déterminées lors de la compilation on utilisera les techniques de vérification de contraintes du paragraphe §9.7.1.

Nous différencions les conteneurs, par la méthode d'accès aux éléments, et par les possibilités d'insertion.

Qu'en est il de nos listes de Cons ?

Puisque ces structures sont déterminées à la compilation cela n'a aucun sens de penser *insérer* un élément, nos listes sont ici dans le même cas que les tableaux et structures, leur type et leur encombrement est connu dès la compilation.

En ce qui concerne l'accès : pour accéder au n-ième èlément il n'y a besoin d'aucun calcul lors de l'exécution puisque son adresse est connue lors de la compilation, le cas la encore est similaire à celui de l'accès à un membre d'une structure.

Mais il nous faut quand même écrire le code qui va *être interprété par le compilateur* pour produire cette adresse. La structure correspondante est donnée 9.6

La structure GetNth est une structure récursive qui fait avancer le compilateur

d'un élément à chaque récursion. La récursion est arrêtée par une *spécialisation* dans le cas de profondeur 1. Dans ce cas il suffit de rendre le `car`. Une fonction utilitaire `getnth` est une fois de plus donnée pour calculer les types et ainsi faciliter l'appel.

Dans cette section nous avons donc vu une manière de créer *à la compilation* des listes *hétérogènes* de taille *fixe* dont le type de chaque élément peut être déduit statiquement par le compilateur. Ces listes dont l'encombrement mémoire est connu à la compilation (voir le programme 9.5) peuvent être stockées sur la pile.

Ces listes ne sont à confondre ni avec les listes homogènes de tailles variables fournies par la STL et décrites §10.10 ni avec les listes hétérogènes d'objets à typage dynamique qui sont communes dans les langages à héritage virtuel (voir §35) et peuvent aussi être réalisées en **C**++ .

Bien qu'inspirées par LISP elles sont encore plus éloignées des listes d'objets non typés de ce langage. Les listes de LISP ont une structure dynamique (on peut remplacer le *car* et le *cdr* d'une liste), admettent le partage, résident en mémoire libre, et ne sont détruites que par un mécanisme de ramasse-miettes.

9.7 Protocole des classes de la STL

9.7.1 Conventions de dénomination.

Comme il serait trop lourd de répéter à chaque utilisation d'un patron les demandes pour chaque argument, une librairie choisira d'associer à un nom particulier des capacités données.

Un nom particulier d'argument patron sera donc associé à un protocole que toute instantiation de cet argument devra posséder.

Ainsi nous pouvons supposer que le nom d'argument patron `InputIterator` correspondra *dans toutes les fonctions patrons de notre bibliothèque* à un argument qui possède les capacités décrites ci-dessus.

Cela constitue une norme de codage propre à la bibliothèque, ces conventions sont non portables, et ne sont pas couvertes par les normes du langage de programmation ; elles peuvent entrer en conflit avec celles proposées par d'autres bibliothèques.

Aussi est-il prudent de se conformer aux conventions suivies dans les bibliothèques standardisées, telles la bibliothèque *C++ standard* ou, pour les patrons, sa partie nommée *Bibliothèque standard de patrons (Standard Template Library)*.

Il est possible de forcer le compilateur à vérifier le protocole d'une classe employée en instanciant des classes qui ne génèrent aucun code mais utilisent les différentes méthodes. On vérifie ainsi un jeux de *contraintes* sur les classes. Des outils ont étés définis pour faciliter la génération et le test de telles contraintes.

Programme 9.7 vérification de contraintes.

```
#include <boost/concept_check.hpp>
template <typename T>
struct IRefable_concept
{
      void constraints() {
         t.incref();
         t.decref();
         i=t.use_count();
      }
      T t;
      int i;
};

template <class T>
class IRef{
      BOOST_CLASS_REQUIRES(T, IRefable_concept);
.....
};
```

La bibliothèque BOOST fournit des outils pour tester les contraintes, ainsi qu'un jeu de tests correspondants aux différentes catégories d'itérateurs de la STL dans l'en-tête `concept-check` qui a été intégré aux versions récentes de la bibliothèque standard.

Un exemple d'utilisation est donnée dans le programme 9.7 qui teste les capacités de l'argument patron de la classe `IRef` du programme 7.12

9.7.2 Les différentes sortes d'itérateurs

Nous décrivons ci-dessous les différentes sortes d'itérateurs, en suivant les conventions de la STL. Ces sortes d'itérateurs sont des classes de bases abstraites de cette bibliothèque.

Tous les classes d'itérateurs définis dans la STL definissent des types associés tels

`value_type` Le type de valeur obtenue en déréférençant l'itérateur.

`distance_type` Le type entier signé servant à représenter une différence d'itérateurs, et donc le nombre d'éléments d'une *portée*.

Tout itérateur ne dérive pas d'une de ces classes de base, car cela interdirait de considérer un pointeurs comme un itérateur, La STL propose donc une classe

patron iterator_traits qui permet de connaître les caractéristiques d'un itérateur que cela soit un pointeur ou un membre d'une classe d'itérateurs de la STL ; elle est définie dans l'en-tête <iterator> [6].

iterator_traits<T> comprend les types suivants :

iterator_category un des types : input_iterator_tag, output_iterator_tag, forward_iterator_tag, bidirectional_iterator_tag ou random_access_iterator_tag qui indiquent la sorte d'itérateur.

value_type Le type de valeur obtenue en déréférençant l'itérateur. Ce type peut être modifiable (*mutable*) quand la valeur obtenue en déréférençant l'itérateur est modifiable, il est constant dans le cas contraire. Souvent un type modifiable peut être en partie gauche d'une affectation il est alors dit affectable *(assignable)*, mais ce n'est pas toujours le cas[7].

Par exemple int* est un itérateur modifiable (*mutable*) alors que const int* est un itérateur constant.

Pour tout type d'itérateur les itérateurs non-valides (par exemple après un delete) et les itérateurs *après la fin* sont non déréférençables. Parmi les itérateurs non déréférençables certaines valeurs sont dites *singulières* ; un itérateur singulier est un itérateur qui n'admet pas les opérations de comparaison usuelles.

Les itérateurs *après la fin* d'un conteneur ne sont pas singuliers, car ils admettent les comparaisons, mais ils sont non déréférençables.

distance_type Le type entier signé servant à représenter une différence d'itérateurs, et donc le nombre d'éléments d'une *portée*.

pointer Un pointeur sur le type de valeur de l'itérateur.

reference Une référence sur le type de valeur de l'itérateur.

Quand nous définissons de nouveaux types d'itérateurs pour les rendre manipulables par la STL ils doivent avoir un iterator_traits, nous avons deux possibilités pour cela : spécialiser iterator_traits, ou plus simplement s'assurer qu'il définit les types requis en dérivant cet itérateur d'une des classes de base input_iterator, output_iterator, forward_iterator, bidirectional_iterator ou random_access_iterator.

[6] La première version de la STL proposait des fonctions surchargées telles value_type ou iterator_category dites *iterator tag functions* cette méthode à été rendue obsolète par les iterator_traits

[7] Une classe peut ne pas définir d'affectation, un pointeur non constant sur la classe sera modifiable sans être affectable.

Trivial Operator : itérateur trivial

Un itérateur trivial est un objet qui peut être déréférencé. La valeur obtenue peut être ou ne pas être modifiable.

Un pointeur sur un objet isolé est un exemple d'itérateur trivial.

Input Iterator : itérateur d'entrée.

Un itérateur d'entrée est un itérateur qui peut être déréférencé et incrémenté, il n'est pas obligé de permettre la modification.

Aprés incrémentation l'ancienne valeur d'un itérateur d'entrée n'est plus valable. Ce comportement restrictif nous permet de les employer pour itérer sur des fichiers conservés sur des support où seule l'information courante est disponible. Ainsi les flots d'entrée possèdent des itérateurs d'entrée.

La classe abstraite correspondante est `input_iterator<T,Distance>` où `T` est le type de valeur et `Distance` est le type (optionnel) de la diffférence d'itérateurs.

Notez que ces classes abstraites sont seulement fournies pour aider à la définition de classes itérateurs conformes aux conventions de la STL.

Pour tout type d'itérateur de la STL le nom du type suivi du mot `_tag` est une classe qui sert d'étiquette pour repérer le type de l'itérateur. La valeur *après la fin* de cet itérateur est obtenue par le constructeur sans argument. Elle nous permet de marquer la fin du flot considéré comme conteneur. Les opérations d'incrémentation sur un itérateur de flot donnent cette valeur à l'itérateur en fin de flot.

Output Iterator : Itérateur de sortie

Les itérateurs de sortie sont des itérateurs qui fournissent un mécanisme pour stocker une suite de valeurs. C'est le pendant des itérateurs d'entrée, mais leur interface est plus réduite : ils ont aussi les deux incrémentations mais pas obligatoirement d'accès à l'élément référencé, ni d'égalité et de différence.

On peut écrire et avancer avec un itérateur de sortie, mais on ne peut pas revenir en arrière, ou au début. On ne peut pas non plus demander la position relative de deux itérateurs de sortie.

Les itérateurs sur un flot de sortie sont des itérateurs de sortie.

La classe abstraite correspondante est `Output_iterator` qui contrairement aux itérateurs d'entrée ne demande pas de type *valeur* ni *différence* puisque les deux opérations correspondantes ne figurent pas au protocole des itérateurs de sortie.

Forward Iterator : itérateur de parcours avant.

Un *Forward Iterator* est un itérateur qui permet un parcours vers l'avant d'une suite de valeurs. Contrairement aux *Input Iterator* et*Output Iterator* une ancienne valeur d'un tel itérateur peut être réemployée.

La classe abstraite correspondante est :

```
forward_iterator<T, Distance>
```

Bidirectional Iterator : Itérateur bidirectionel.

Un itérateur bidirectionnel a les mêmes propriétés qu'un itérateur avant, auxquelles sont ajoutées les incrémentations préfixes et postfixes. Un pointeur sur un tableau est un exemple d'itérateur bidirectionnel. Les listes avec un double chainage fournissent aussi des itérateurs bidirectionnels.

La classe abstraite correspondante est :

```
bidirectional_iterator<T, Distance>
```

elle permet l'emploi des fonctions surchargées `iterator_category`, `distance_type`, et `value_type`.

Random Access Iterator : Itérateur d'accès direct.

Un itérateur d'accès direct est un itérateur bidirectionnel qui peut aussi effectuer des déplacements par saut vers l'avant et l'arrière en temps constant. Les opérations d'addition et de soustraction d'un entier (de type `Distance`) sont définies pour un itérateur d'accès direct. Pour un tel itérateur `it += n` a le même effet que l'application répétée *n* fois de `it++`. L'opération `it[n]` est aussi définie et équivalente à `*(it + n)`.

Les pointeurs sont des exemples d'itérateurs d'accès direct ainsi que les itérateurs des classes `vector` et `deque` de la STL.

La classe abstraite correspondante est :

```
random_access_iterator<T, Distance>
```

Chapitre 10

La bibliothèque standard de patrons

10.1 Le conteneur `vector`

`vector<T,Alloc>`

Le conteneur le plus utilisé de la bibliothèque standard se nomme `vector`.Il est destiné à contenir une suite d'éléments en permettant un accès direct à tout élément et une insertion ou suppression rapide en fin de séquence. Pour l'utiliser on inclura l'en-tête `<vector>` elle est inclue dans l'espace de nom `std`.

10.1.1 Types et accesseurs

Types

La classe `vector` comme les autres classes de la STL définit et exporte plusieurs types :

`value_type` *(Container)*

Le type d'élément stocké dans le conteneur. C'est le type rendu par les accesseurs.

`reference`, et `const_reference` *(Container)*)

Le type d'une référence (resp. référence constante) à un élément du conteneur.

`difference_type` *(Container)*

Un type entier signé utilisé pour les différences d'itérateurs. Il permet de compter le nombre d'objets qui séparent deux éléments du conteneur.

`size_type` *(Container)*

Un type entier non signé, qui permet de représenter toute valeur positive de `difference_type`. différence d'itérateurs.

Itérateurs

`iterator`, et `const_iterator` *(Container)*

Ces types d'itérateurs permettent de traverser le conteneur. Pour les vecteurs il s'agit d'itérateurs à accès direct.

Un `const_iterator` permet de consulter les éléments du conteneur, mais non de les modifier. Il peut s'appliquer à un conteneur constant.

`begin()` et `end()` sont deux itérateurs situés au début et après la fin du vecteur. Pour des vecteurs constants, ces méthodes par surcharge rendent des `const_iterator`.

`reverse_iterator` et `const_reverse_iterator` *(Reversible Container)*

Ces types sont similaires à `iterator` mais qui permettent de traverser le conteneur de la fin vers le début.

`reverse_iterator rbegin()` et `reverse_iterator rend()` *(Reversible Container)*
délivrent des itérateurs situés au début et après la fin du vecteur *pour une traversée de la fin au début*. [1]

Références au début et fin de conteneur.

Les accesseurs `back` et `front` permettent d'utiliser le vecteur comme une pile.

`reference front()`, `const_reference front() const` *(Sequence)*
retournent une référence (respectivement référence constante) au premier élément et sont équivalentes à `*(a.first())`.

`reference back()`, et `const_reference back() const` *(Back Insertion Sequence)*
retournent une référence au dernier élément et sont équivalents à `*(--a.end())`.

---

[1] `rbegin()` est différent de `end()` car ils n'ont pas le même type et que de plus `end()` est situé *après* le dernier élément et ne permet donc pas d'accéder à un élément valide du vecteur.

Accès direct à un élément

Pour accéder de manière directe à un élément nous disposons de quatre méthodes.

`reference operator[](size_type n)` *(Random Access Container)*
: permet d'accéder à un vecteur de la même manière qu'à un tableau **C**. [2]

`const_reference operator[](size_type n)const` *(Random Access Container)*
: est l'opérateur constant correspondant.

`reference at(size_type n)`*(Random Access Container)* et `const_reference at(size_type n) const`
: sont des méthodes similaires à l' un opérateur d'indexation `[]` mais elles vérifient l'indice et lancent une exception `std::out_of_range` si celui-ci n'est pas valide.

10.1.2 Construction

La classe vecteur à les constructeurs suivants :

`vector()` *(Container)*

Il construit un nouveau vecteur *vide*, c'est-à-dire de longueur 0 et s'utilise sous la forme `vector<T>v`.

`vector(size_type n)` *(Sequence)*
construit un vecteur de longueur initiale *n* initialisé avec le constructeur par défaut de `T`.

On l'utilise dans une définition : `vector<T>v(n)`

`vector(size_type n,const T& t)` *(Sequence)*
construit un vecteur en précisant une valeur initiale spécifique pour les `n` membres.

On l'utilise dans une définition : `vector<T>v(n,valeur)`

`vector(const vector&)` *(Container)*
est le constructeur de copie.

On l'utilise dans une définition : `vector<T>v2(v1)`

---

[2] voir aussi §29.

Programme 10.1 Remplissage d'un vecteur vide.

```
#include <iostream>
#include <vector>
using std::cout; using std::endl;
using std::vector;

int main ()
{
  vector<int> v1; // vecteur vide
  cout << "empty = " << v1.empty () << endl;
  cout << "size = " << v1.size () << endl;
  cout << "max_size = " << v1.max_size () << endl;
  v1.push_back (42); // Ajoute un entier au vecteur
  cout << "size = " << v1.size () << endl;
  cout << "v1[0] = " << v1[0] << endl;
  return 0;
}
```

Résultat

```
empty = 1
size = 0
max_size = 1073741823
size = 1
v1[0] = 42
```

`vector(iterator premier,iterator dernier)` *(Sequence)*

C'est un constructeur qui initialise le nouveau vecteur avec les éléments présents entre deux itérateurs, c'est-à-dire ce que nous appelons une *portée* ou un intervalle.

La classe `vector` a aussi un constructeur de copie qui permet de créer un vecteur avec le contenu d'un vecteur existant :

```
vecteur<T> v2(v);
```

Toutes ces constructions se font par *copie* des objets et reposent donc sur la validité du constructeur de copie de la classe des éléments. Ce constructeur de copie est aussi utilisé pour réallouer le vecteur quand sa taille devient insuffisante. Notons que tout conteneur *possède* ses éléments dont la durée de vie ne peut excéder celle du conteneur, mais les éléments peuvent être des formes de pointeurs qui pointent sur des objets qui durent plus longtemps que le pointeur.

Enfin les vecteurs ont un destructeur :

Programme 10.2 initialisation avec une copie d'un vecteur.

```
#include <iostream>
#include <vector>
using std::cout; using std::endl;
using std::vector;

int main ()
{
  vector<char> v1; // vecteur  de caractères vide.
  v1.push_back ('h');
  v1.push_back ('i');
  cout << "v1 = " << v1[0] << v1[1] << endl;
  vector<char> v2 (v1); //copie de v1
  v2[1] = 'o'; // Remplace la seconde entrée
  cout << "v2 = " << v2[0] << v2[1] << endl;
  cout << "(v1 == v2) = " << (v1 == v2) << endl;
  cout << "(v1 < v2) = " << (v1 < v2) << endl;
  return 0;
}
```

Résultat

```
v1 = hi
v2 = ho
(v1 == v2) = 0
(v1 < v2) = 1
```

`~vector()` qui détruit tous les éléments et récupère la place qu'ils occupaient.

Dans la destruction d'un conteneur les *destructeurs* des éléments sont appelés. Nous devons donc prendre garde à une destruction de vecteurs de pointeurs qui ne détruirait *pas* les objets pointés. On préférera souvent un vecteur d'une classe de *pointeurs intelligents*, car ceux-ci savent gérer l'allocation mémoire des objets et la recopie des pointeurs.

Les programmes 10.1, 10.2 et 10.3 donnent des exemples de constructions de vecteurs.

10.1.3 Comparaison et affectation

La bibliothèque standard définit l'égalité et l'affectation pour deux conteneurs quelconques ayant le même type de base.

Deux conteneurs sont considérés comme égaux s'ils ont la même taille et que

Programme 10.3 Inititialisation d'un vecteur avec une portée.

```
#include <iostream>
#include <vector>
using std::vector;

  int array [] = { 1, 4, 9, 16 };

  int main () {
  vector<int> v (array, array + 4);
  for (int i = 0; i < v.size (); i++)
  std::cout << "v[" << i << "] = " << v[i] <<  std::endl;
  return 0;
}
```

Résultat

```
v[0] = 1
v[1] = 4
v[2] = 9
v[3] = 16
```

les éléments de même position sont égaux d'après l'opérateur `==`.

De la même manière des comparaisons suivant l'ordre lexicographique sont définies pour deux conteneurs de même type et associées à l'opérateur `<`.

Pour deux conteneurs `x` et `y` de même type de base l'affectation `x=y` est définie et elle remplace les éléments de `x` par ceux de `y`.

Enfin la fonction membre `swap` échange les valeurs du conteneur avec celles d'un autre conteneur, après `x.swap(y)` les valeurs de `x` et `y` sont interverties.

Le programme 10.4 donne un exemple d'affectation et d'échange du contenu de deux vecteurs.

Les prototypes de ces fonctions *globales* [3] sont :

`bool operator==(const vector&,const vector&)` *(Forward Container*

`bool operator<(const vector&,const vector&)` *(Forward Container*

et celle de la méthode `swap` :

`void swap(vector&)` *(Container)*

10.1.4 Insertion et suppression d'éléments

La taille du vecteur est donnée par la méthode :

[3] ce ne sont pas des méthodes.

Programme 10.4 Affectation et échange de vecteurs.

```
#include <iostream>
#include <vector>
using std::vector;

void print (vector<double>& vec)
{
  for (unsigned i = 0; i < vec.size (); i++)
    std::cout << vec[i] << " ";
  std::cout << std::endl;
}

int main ()
{
   using std::cout;
  vector<double> v1; // vecteur vide de doubles.
  v1.push_back (32.1);
  v1.push_back (40.5);
  vector<double> v2; // autre vecteur vide de  doubles.
  v2.push_back (3.56);
  cout << "v1 = ";
  print (v1);
  cout << "v2 = ";
  print (v2);
  v1.swap (v2); // échange des contenus.
  cout << "v1 = ";
  print (v1);
  cout << "v2 = ";
  print (v2);
  v2 = v1; // affectation.
  cout << "v2 = ";
  print (v2);
  return 0;
}
```

Résultat

```
v1 = 32.1 40.5
v2 = 3.56
v1 = 3.56
v2 = 32.1 40.5
v2 = 3.56
```

`size_type size() const` *(Sequence)*

elle est toujours inférieure à la taille maximale d'un vecteur qui est :

`size_type max_size() const` *(Sequence)*

La taille réservée en mémoire, qui est toujours supérieure à `size()` est :

`size_type capacity() const` *(vector)*

Le gestionnaire mémoire gère la place occupée par le vecteur et prend entièrement en charge `capacity`. Il alloue la place mémoire par tranches de manière à limiter le nombre de réallocations et de recopies lors de l'extension du vecteur. Toutes les réallocations mémoires invalident tous les itérateurs sur des éléments du vecteur.

Si on connaît par avance la taille maximale que va prendre le vecteur on peut soit réserver la place lors de la construction du vecteur si c'est une quantité statique ou alors demander de réserver de la place avec :

`void reserve(size_type n)` *(vector)*

qui augmente la capacité à un chiffre *supérieur* à n si cela est possible. Dans tous les cas la taille `size()` est inchangée.

L'utilisation de `reserve` permet aussi de contrôler la validité des itérateurs en limitant les réallocations mémoires.

Deux méthodes efficaces, car en temps constant, permettent d'insérer et de retirer un élément en fin de vecteur, ce sont :

`void push_back(const T&)` *(Back Insertion Sequence)*

`void pop_back()` *(Back Insertion Sequence)*

Le programme 10.5 utilise les méthodes `front()` et `back()` qui retournent une référence aux premier et dernier éléments.

L'insertion et la suppression en milieu de vecteur se font par :

`iterator insert(iterator pos,const T& x)` *(Sequence)*

`iterator erase(iterator pos)` *(Sequence)*

Des versions pour insérer en une fois une série d'éléments, plus efficaces que des opérations répétées, sont aussi prévues :

```
Template <class InputIterator>
    void insert(iterator pos,InputIterator f,InputIterator l)
```
(Sequence)

`iterator erase(iterator first,iterator last)` *(Sequence)*

L'insertion et la suppression en milieu de vecteur sont très pénalisantes car il faut recopier tous les éléments après celui qui est ajouté ou supprimé. Elles sont d'un temps linéaire, i.e. proportionelle à la taille du vecteur et non plus constant.

Quand de nombreuses insertions ou suppressions en milieu d'un conteneur sont prévues on optera pour un type de conteneur mieux adapté tel que le type

Programme 10.5 Utilisation de `front` et `back` dans un vecteur.

```
#include <iostream>
#include <vector>

int main ()
{
  using std::vector;
  using std::cout; using std::endl;
  vector<int> v (4);
  v[0] = 1;
  v[1] = 4;
  v[2] = 9;
  v[3] = 16;
  cout<<"front = "<<v.front()<<endl;
  cout<<"back = "<<v.back()<<", size = "<<v.size()<<endl;
  v.push_back (25);
  cout<<"back = "<<v.back()<<", size = "<<v.size()<< endl;
  v.pop_back ();
  cout<<"back = "<<v.back()<<", size = "<<v.size()<< endl;
  return 0;
}
```

Résultat

```
front = 1
back = 16, size = 4
back = 25, size = 5
back = 16, size = 4
```

`List`. Quand seules une insertion et suppression en début de conteneur sont nécessaires on optera pour un `deque`.

Il est important de se souvenir que ces opérations causent des réallocations mémoire et des recopies et que *tous les itérateurs sur un vecteur sont invalidés après une réallocation mémoire* .

Même sans réallocation, tous les itérateurs sur une position située après celle d'une insertion ou d'une suppression sont invalidés.

10.2 Algorithmes

La bibliothèque des patrons fournit un grand nombre d'algorithmes dans l'entête `<algorithm>`, qui permettent de traiter les conteneurs. Ces algorithmes uti-

Programme 10.6 Insertion et effacement dans un vecteur.

```
#include <iostream>
#include <vector>
using std::vector;

int array1 [] = { 1, 4, 25 };
int array2 [] = { 9, 16 };

int main ()
{
  using std::cout; using std::endl;
  vector<int> v (array1, array1 + 3);
  vector<int>::size_type i;
  v.insert (v.begin (), 0);// Insertion avant le premier élement.
  v.insert (v.end (), 36); // Insertion après le dernier élément.
  for ( i = 0; i < v.size (); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;
  // Insertion d'un tableau avant le quatrième élément.
  v.insert (v.begin () + 3, array2, array2 + 2);
  for (i = 0; i < v.size (); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;
  // Efface tout sauf le premier et le dernier.
  v.erase (v.begin () + 1, v.end () - 1);
  for (i = 0; i < v.size (); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;
  return 0;
}
```

Résultat

```
v[0] = 0
v[1] = 1
v[2] = 4
v[3] = 25
v[4] = 36

v[0] = 0
v[1] = 1
v[2] = 4
v[3] = 9
v[4] = 16
v[5] = 25
v[6] = 36

v[0] = 0
v[1] = 36
```

Programme 10.7 Utilisation de l'algorithme `find` pour un tableau.

```
#include <algorithm>
#include <iostream>

const int nombres[] = { 0, 1, 4, 9, 16, 25, 36, 49, 64 };

const int* const finnombres=nombres+sizeof(nombres)/sizeof(int);

int main ()
{
  const int* place (std::find (nombres, finnombres, 25));
  std::cout << "25 trouvé à l'indice: " << (place - nombres) << std::endl;
  return 0;
}
```

Résultat

```
25 trouvé à l'indice: 5
```

lisent les conteneurs par le biais des itérateurs, ils sont donc indépendants de la structure des conteneurs et peuvent travailler sur tous ceux qui fournissent les itérateurs nécessaires.

Ces algorithmes sont répartis en deux groupes, d'une part ceux qui modifient le contenu du conteneur traité, d'autre part ceux qui ne le modifient pas et délivrent éventuellement un autre conteneur en résultat.

10.2.1 Algorithmes ne modifiant pas le conteneur.

Exemples détaillés

recherche de la valeur d'un élément : `find`.

```
template <class InputIterator, class T>
InputIterator find(InputIterator first,
                   InputIterator last, const T& value);
```

La fonction patron `find` renvoie un itérateur sur le premier élément égal à `value` entre les éléments associés à `first` et `last`.

Un exemple qui utilise un tableau est donné dans le programme 10.7 un autre utilisant les vecteurs est donné programme 10.8 et dans le programme 1.10 page 10.

Programme 10.8 Utilisation de l'algorithme `find` pour un vecteur.

```
#include <algorithm>
#include <iostream>
#include <vector>

typedef std::vector<char> Chaine;
typedef Chaine::iterator ChaineIndex;
char t[]="un exemple de texte.";
Chaine vt(t,t+strlen(t));

int main()
{
   ChaineIndex place1 ( std::find (vt.begin(),vt.end(),' '));
   ChaineIndex place2 ( std::find (place1+1, vt.end(),' '));
   std::cout <<"mot de longueur " << place2-place1-1;
   std::cout << " à l'indice: "<< place1-vt.begin()+1 << std::endl;
}
```

Résultat

```
mot de longueur 7 à l'indice : 3
```

La puissance de l'outil *Patron* est illustrée par la simplicité de l'algorithme de `find` :

```
template <class InputIterator, class T>
InputIterator find(InputIterator first,
                   InputIterator last, const T& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

Notons que nous n'utiliseront pas l'algorithme `find` pour les tableaux associatifs, mais la fonction membre `find`, qui tire parti de la structure du conteneur pour effectuer un accès direct.

Recherche d'un élément vérifiant une condition : `find_if`.

```
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

Programme 10.9 Recherche avec `find_if`

```
#include <algorithm>
#include <iostream>
#include <vector>

bool div3 (int a) {return a % 3 == 0;}

int main ()
{
  using std::cout;
  typedef std::vector <int> IntVec;
  IntVec v (10);
  for (unsigned i = 0; i < v.size (); i++)
    v[i] = (i + 1) * (i + 1);
  IntVec::iterator iter;
  iter = find_if (v.begin (), v.end (), div3);
  if (iter != v.end ()){
    cout  << "La valeur " << *iter;
    cout  << " à l'indice: " << (iter - v.begin ());
    cout  << " est divisible par 3" << std::endl;
  }else{
     cout  << "aucune valeur n'est divisible par 3"<<std::endl;
  }
  return 0;
}
```

Résultat

```
La valeur 9 à l'indice: 2 est divisible par 3
```

L'argument `pred` est un prédicat c'est-à-dire une fonction ou un objet-fonction renvoyant une valeur de type `bool`. `find_if` renvoie le premier élément de l'intervalle qui vérifie le prédicat `pred`. Le programme 10.9 illustre une recherche dans un vecteur d'entiers du premier nombre divisible par trois.

Appliquer une fonction à chaque élément : `for_each`

```
template <class InputIterator, class Function>
Function for_each(InputIterator first,
                  InputIterator last, Function f)
```

`f` doit être une fonction ou un objet-fonction unaire ; il est appliqué à tous les éléments de l'intervalle `[first,last)`. Le programme 10.10 illustre l'utilisation

Programme 10.10 Utilisation de `for_each` sur tous les éléments d'un vecteur.

```
#include <algorithm>
#include <iostream>
#include <vector>

void ecritCarre (int a) { std::cout << a * a << " ";}

int main ()
{
  std::vector <int> v1 (10);
  for (unsigned int i = 0; i < v1.size (); i++){
    v1[i] = i;
  }
  for_each (v1.begin (), v1.end (), ecritCarre);
  std::cout << std::endl;
  return 0;
}
```

Résultat

```
0 1 4 9 16 25 36 49 64 81
```

de `for_each` pour effectuer un traitement sur tous les éléments d'un conteneur.

`for_each` retourne la valeur de l'objet fonction après l'itération : c'est utile dans le cas où celui-ci modifie son état à chaque itération, comme l'illustre l'exemple d'un additionneur dans le programme 10.11.

Autres algorithmes

Nous donnons ci-dessous un résumé de quelque autres algorithmes ne modifiant pas le conteneur, les déclarations `Template` ne sont pas rappelées.

Sélection de valeurs

À côté de `find` et `find_if`, que nous avons déjà vus, on trouve encore :

```
iterator_traits<InputIterator>::difference_type
    count(InputIterator first,InputIterator last,
        const EqualityComparable& value);
```

compte le nombre d'éléments égaux à `value` et rend ce nombre dont le type est `iterator_traits<InputIterator>::difference_type`. La classe `iterator_traits` sert seulement à sélectionner le bon type de retour lors de

Programme 10.11 Itération avec un additionneur.

```
#include <iostream>
#include <functional>
#include <numeric>

class Additionneur : public std::unary_function<double, void>
{
   public:
      Additionneur() : somme(0) {}
      void operator()(double x) { somme += x;}
      double total() { return somme;}
   private:
      double somme;
};

int main(){
   std::vector<double> v(10);
   std::iota(v.begin(),v.end(),0);
   std::copy(v.begin(), v.end(),
             std::ostream_iterator<double>(std::cout, " "));
   std::cout << std::endl;
   Additionneur result=for_each(v.begin(), v.end(), Additionneur());
   std::cout << "total: " << result.total() << std::endl;
   std::cout << "somme par accumulate ";
   std::cout << std::accumulate(v.begin(), v.end(),0.)<<std::endl;
}
```

Résultat

```
0 1 2 3 4 5 6 7 8 9
total: 45
somme par accumulate 45
```

l'instantiation. Dans les anciennes versions elle n'est pas disponible et on utilise :

```
void count(InputIterator first,InputIterator last,
           const EqualityComparable& value,
           Size& n);
```

où `n` contient le résultat.

```
iterator_traits<InputIterator>::difference_type
  count_if(InputIterator first,InputIterator last,
           Predicate pred)
```

délivre un itérateur sur la première sous-séquence de `[first1,last1[` qui est identique à `[first,last)` ; si aucune séquence n'est trouvée `last1` est rendu.

La première fonction compare avec `==`, la seconde avec `binary_pred`.

```
ForwardIterator1
  find_end(ForwardIterator1 first1,ForwardIterator1 last1,
           ForwardIterator2 first2,ForwardIterator2 last2);
ForwardIterator1
  find_end(ForwardIterator1 first1,ForwardIterator1 last1,
           ForwardIterator2 first2,ForwardIterator2 last2,
           BinaryPredicate comp);
```

sont similaires à `search` mais la *dernière* séquence et non la première est cherchée. `last1` est rendu si aucune correspondance n'est trouvée.

```
ForwardIterator
  max_element(ForwardIterator first,ForwardIterator last);
```

délivre un itérateur sur le plus grand élément de `[first,last)`.

```
ForwardIterator
  max_element(ForwardIterator first,ForwardIterator last,
              BinaryPredicate comp);
```

est similaire mais utilise `comp` pour comparer les éléments.

```
ForwardIterator min_element(ForwardIterator first,
                            ForwardIterator last);
```

délivre un itérateur sur le plus petit élément de `[first,last)`.

```
ForwardIterator
  min_element(ForwardIterator first,ForwardIterator last,
              BinaryPredicate comp);
```

est similaire mais utilise `comp` à la place de `<` pour comparer les éléments.

```
ForwardIterator
  adjacent_find(ForwardIterator first,ForwardIterator last);
```

délivre un itérateur sur le premier élément qui est immédiatement suivi par un élément qui lui est égal.

```
ForwardIterator
  adjacent_find(ForwardIterator first,ForwardIterator last,
                BinaryPredicate binary_pred);
```

utilise `binary_pred` à la place de l'égalité.

```
ForwardIterator lower_bound(
      ForwardIterator first,ForwardIterator last,
      const LessThanComparable& value);
```

effectue une *recherche binaire* dans une portée *ordonnée*, `[first,last)` il retourne le plus grand itérateur `prem` tel que tous les élements de `[first,prem)` soient strictement inférieurs à `value`. Cela correspond à la plus petite position où on peut insérer l'élément en respectant l'ordre.

Un exemple d'utilisation est donné dans le prrogramme 1.10 page 10.

```
ForwardIterator lower_bound(
      ForwardIterator first,ForwardIterator last,
      const T& value,StrictWeakOrdering comp);
```

fonctionne de manière similaire mais les comparaisons se font à l'aide de `comp` à la place de `<`.

```
ForwardIterator upper_bound(
      ForwardIterator first,ForwardIterator last,
      const LessThanComparable& value);
```

effectue une *recherche binaire* dans une portée *ordonnée* `[first,last)`. Il retourne le plus grand itérateur `prem` tel qu'aucun élements de `[first,prem)` ne soit strictement supérieur à `value` ; cela correspond à la plus grande position ou on peut insérer l'élément en respectant l'ordre.

```
ForwardIterator upper_bound(
      ForwardIterator first,ForwardIterator last,
      const T& value,StrictWeakOrdering comp);
```

fonctionne de manière similaire mais les comparaisons se font à l'aide de `comp` à la place de `<`.

```
bool binary_search(ForwardIterator first,ForwardIterator last
                   const LessThanComparable& value);
```

```
bool binary_search(ForwardIterator first,ForwardIterator last
                   const T& value,StrictWeakOrdering comp);
```

effectuent une *recherche binaire* dans une portée *ordonnée* `[first,last)` et retournent le premier itérateur correspondant à un élément équivalent à `value`. Deux éléments sont équivalents quand aucun n'est plus petit que l'autre.

Le premier algorithme utilise `<` pour la comparaison, le second `comp`.

Accumulation de valeurs

```
T accumulate(InputIterator first,InputIterator last,T init);
```

fait la somme de `init` et de tous les objets de `[first,last)`.

```
T accumulate(InputIterator first,InputIterator last,T init,
           BinaryFunction binary_op);
```

est similaire mais applique `binary_op` au lieu de `+`.

```
T inner_product(InputIterator1 first1,InputIterator1 last1,
                InputIterator2 first2,T init)
```

réalise la somme de `init` et des produits d'éléments de même rang des deux conteneurs.

```
T inner_product(InputIterator1 first1,InputIterator1 last1,
      InputIterator2 first2,T init,BinaryFunction1 binary_op1,
      BinaryFunction2 binary_op2)
```

est identique mais la somme est remplacée par `binary_op1` et le produit par `binary_op2`

```
OutputIterator
  partial_sum(InputIterator first,InputIterator last,
```

```
              OutputIterator result);
```

est telle que le conteneur parcouru par `résult` contiendra dans son n-ième élément la somme des n premiers éléments de `[first,last)`.

```
OutputIterator
  partial_sum(InputIterator first,InputIterator last,
              OutputIterator result,BinaryOperation binary_op);
```

remplace la somme par `binary_op`.

```
OutputIterator
  adjacent_difference(InputIterator first,InputIterator last,
                      OutputIterator result);
```

met dans le n-ième élément du conteneur parcouru par `résult` la différence du n+1 ième et du n-iéme élément de `[first,last)`.

```
OutputIterator
  adjacent_difference(InputIterator first,InputIterator last,
              OutputIterator result,BinaryFunction binary_op);
```

remplace la différence par `binary_op`.

Comparaison de deux conteneurs

```
bool equal(InputIterator1 first1,InputIterator1 last1,
           InputIterator2 first2);
```

retourne *vrai* si les deux conteneurs sont égaux.

```
bool equal(InputIterator1 first1,InputIterator1 last1,
           InputIterator2 first2,BinaryPredicate binary_pred);
```

est identique à la fonction précédente mais l'égalité est remplacée par `binary_pred`

```
pair<InputIterator1,InputIterator2>
  mismatch(InputIterator1 first1,InputIterator1 last1,
           InputIterator2 first2);
```

délivre les itérateurs pointant sur la première paire d'éléments distincts des deux conteneurs.

```
pair<InputIterator1,InputIterator2>
  mismatch(InputIterator1 first1,InputIterator1 last1,
           InputIterator2 first2,BinaryPredicate binary_pred);
```

est similaire mais utilise `binary_pred` à la place de l'égalité.

```
bool lexicographical_compare(
                   InputIterator1 first1,InputIterator1 last1,
                   InputIterator2 first2,InputIterator2 last2);
```

retourne la comparaison lexicographique de `[first1,last1)` et `[first2, last2)`.

```
bool lexicographical_compare(
    InputIterator1 first1,InputIterator1 last1,
    InputIterator2 first2,InputIterator2 last2,
    BinaryPredicate comp);
```

effectue une comparaison lexicographique avec la relation d'ordre `comp` à la place de `<`.

```
template <class InputIterator1,class InputIterator2>
int lexicographical_compare_3way(
   InputIterator1 first1,InputIterator1 last1,
   InputIterator2 first2,InputIterator2 last2);
```

effectue comparaison lexicographique de `[first1,last1)` et `[first2,last2)` et retourne -1, 0 ou $+1$ suivant que la première portée est respectivement plus petite, égale, ou plus grande que la seconde.

```
ForwardIterator min_element(
  ForwardIterator first,ForwardIterator last);
ForwardIterator min_element(
  ForwardIterator first,ForwardIterator last,
  BinaryPredicate comp);
const T& min(const T& a,const T& b);
const T& min(const T& a,const T& b,BinaryPredicate comp);
```

cherchent des minima en utilisant l'opérateur `<` ou le prédicat `comp`. Ils retournent le premier élément qui n'est dominé par aucun autre élément de la portée.

10.2.2 Algorithmes modifiant le conteneur.

Exemples détaillés

Appliquer une fonction aux éléments d'un conteneur : `transform`.

Cet algorithme peut être utilisé pour appliquer une fonction à chaque élément d'un conteneur et le *modifier* en remplaçant les anciens éléments par les nouveaux éléments, il peut aussi être utilisé pour stocker le résultat dans un autre conteneur en ne modifiant pas l'original.

Sa syntaxe est :

```
OutputIterator transform(
          InputIterator first, InputIterator last,
          OutputIterator result, UnaryFunction op);
```

Une autre version applique une opération binaire aux éléments correspondants de deux conteneurs :

```
OutputIterator transform(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, OutputIterator result,
        BinaryFunction binary_op);
```

`result` peut être identique à `first` si on effectue la transformation sur place, mais il ne peut pas pointer sur un autre élément du conteneur d'origine. Le conteneur résultat doit être suffisamment grand pour contenir le résultat.

Le programme 10.12 donne exemple d'utilisation en employant une fonction unaire qui calcule l'opposé d'un nombre ; le programme 10.13 utilise en revanche une opération binaire et produit le remplacement du premier conteneur par le résultat.

Copier des éléments d'un intervalle vers un autre : `copy`.

```
template <class InputIterator, class OutputIterator>
  OutputIterator copy(InputIterator first, InputIterator last,
                      OutputIterator result);
```

Il s'agit d'une copie vers l'avant de l'intervalle `[first,last)` à partir de `result`. L'itérateur `result` ne doit pas appartenir à `[first,last)` et doit pointer sur un conteneur de taille suffisante.

Le programme 10.14 illustre une recopie d'un vecteur dans un vecteur plus grand.

Programme 10.12 Utilisation de `transform` sur un vecteur.

```
#include <iostream>
#include <vector>
#include <algorithm>

int oppose (int a){  return -a;}
void ecrit(int a){std::cout << a << " ";}

int nombres[6] = { -5, -1, 0, 1, 6, 11 };

int main ()
{
  std::vector<int> resultat(6);
  std::transform (nombres, nombres + 6, resultat.begin(), oppose);
  std::for_each (resultat.begin(), resultat.end(), ecrit);
  std::cout << std::endl;
  return 0;
}
```

Résultat

```
5 1 0 -1 -6 -11
```

Copie à rebours : `copy_backward`.

Quand les itérateurs peuvent aussi se déplacer vers l'arrière, c'est-à-dire sont des `Bidirectional Iterator`s nous avons aussi une version qui effectue la copie à rebours :

```
BidirectionalIterator2 copy_backward(
       BidirectionalIterator1 first, BidirectionalIterator1 last,
       BidirectionalIterator2 result);
```

La encore `result` ne doit pas appartenir à `[first,last)` et pointer *à la fin* d'un intervalle de taille suffisante. Cependant les intervalles peuvent se chevaucher.

Le programme 10.15 montre une copie arrière avec des intervalles non disjoints.

Autres algorithmes

Suppression d'éléments.

```
ForwardIterator remove(ForwardIterator first,
```

Programme 10.13 Utilisation de `transform` avec une fonction binaire.

```
#include <iostream>
#include <algorithm>
#include <cstring>

char map_char (char a, int b)
{
  return char(a + b);
}

int trans[] = {-8, 0, -11, 9, -16, -88, 65, 0, -1, 1, -7};
char n[] = "Joyeux noèl";

int main ()
{
  const unsigned nbchar = std::strlen (n);
  std::transform (n, n + nbchar, trans, n, map_char);
  std::cout << n << std::endl;
  return 0;
}
```

Résultat

```
Bonne année
```

```
        ForwardIterator last,const T& value);
```

change le conteneur et son itérateur de fin `last`, de manière à ce que `[first, new-last)` ne contienne plus d'élément égal à `value` et que ces éléments soient regroupés dans `[new-last,last)`.

Contrairement à ce que peut suggérer son nom cet algorithme n'effectue aucune suppression du conteneur, celles-ci doivent être effectuée ensuite, si nécessaire.

```
ForwardIterator remove_if(ForwardIterator first,
        ForwardIterator last,Predicate pred);
```

supprime les éléments vérifiant une condition.

`last` est modifié pour que l'intervalle ne contienne plus d'éléments vérifiant le prédicat.

```
ForwardIterator unique(ForwardIterator first,
        ForwardIterator last);
```

Programme 10.14 Utilisation d'un algorithme de copie

```
#include <iostream>
#include <algorithm>
using std::vector;

vector<int> s1;
vector<int> s2(20);

int main(){
   for (int i=0; i<10; ++i){
      s1.push_back(i);
   }
  copy (s1.begin(),s1.end(),s2.begin());
  for (vector<int>::iterator it(s1.begin()); it < s1.end(); ++it){
     std::cout << *it << " ";
  }
  std::cout << std::endl;
}
```

Résultat

```
0 1 2 3 4 5 6 7 8 9
```

Programme 10.15 Utilisation d'un algorithme de copie arrière

```
#include <iostream>
#include <algorithm>

char s [] = "joyeux noël!!!!!!!!";

int main(){
  std::copy_backward( s, s+11, s+strlen(s)-1);
  std::cout << s << std::endl ;
}
```

Résultat

```
joyeux joyeux noël!
```

modifie `last` pour éliminer les éléments dupliqués de l'intervalle `[first, last)`.

```
ForwardIterator unique(ForwardIterator first,
        ForwardIterator last,BinaryPredicate binary_pred);
```

est identique mais effectue la comparaison avec `binary_pred`.

Copie d'éléments

```
OutputIterator unique_copy(InputIterator first,
        InputIterator last,OutputIterator result);
```

fonctionne comme `copy` mais copie un seul exemplaire d'un groupe d'éléments consécutifs égaux.

```
OutputIterator unique_copy(InputIterator first,
        InputIterator last,OutputIterator result,
        BinaryPredicate binary_pred);
```

est identique au précédent mais mais effectue la comparaison avec `binary-_pred`.

```
OutputIterator remove_copy(InputIterator first,
        InputIterator last,OutputIterator result,
        const T& value);
```

copie les éléments *différents* de `value`.

```
OutputIterator
  remove_copy_if(InputIterator first,InputIterator last,
                 OutputIterator result,Predicate pred);
```

copie les éléments qui ne *vérifient pas* `pred`.

```
OutputIterator reverse_copy(BidirectionalIterator first,
        BidirectionalIterator last,OutputIterator result);
```

copie une image miroir de l'intervalle `[first,last)` à partir de `result`. Un itérateur sur la fin du second intervalle est rendu.

```
void reverse(BidirectionalIterator first,
        BidirectionalIterator last);
```

remplace `[first,last)` par son image miroir.

Remplissage de conteneurs

```
void fill(ForwardIterator first,ForwardIterator last,
         const T&value);
OutputIterator fill_n(OutputIterator first,Size n,
         const T& value);
```

affectent la valeur `value` à tous les éléments de l'intervalle `[first,last)` ou respectivement à `n` éléments à partir de `first`.

```
void iota(ForwardIterator first,
          ForwardIterator last,T value);
```

remplit le conteneur avec les valeurs successives obtenues en incrémentant la valeur entrée par `value++`.

```
void generate(ForwardIterator first,ForwardIterator last,
              Generator gen);
OutputIterator generate_n(OutputIterator first,Size n,
                          Generator gen);
```

appellent l'objet-fonction sans argument `gen`, et affectent le résultat successivement à tous les éléments de l'intervalle `[first,last)`, ou respectivement à `n` éléments à partir de `first`.

Construction

Les algorithmes suivants servent à effectuer des constructions, dans une mémoire déjà allouée. Leur utilisation reste exceptionnelle, l'allocation et l'initialisation étant de préférence groupées en une seule opération.

```
void construct(T1* p,const T2& value);
```

construit tous les objets de l'intervalle `[first,last)`.

```
ForwardIterator uninitialized_fill_n(
        ForwardIterator first,Size n,const T& x);
```

est semblable mais construit `n` éléments.

```
ForwardIterator uninitialized_copy(InputIterator first,
        InputIterator last,ForwardIterator result);
```

effectue une *construction* dans un conteneur dont les élément ne sont pas encore initialisés. Cette copie *n'est pas effectuée avec l'opérateur de copie*.

```
void destroy(ForwardIterator first,ForwardIterator last);
```

appelle les destructeurs sans récupérer la mémoire.

Tris et fusions.

```
void sort(RandomAccessIterator first,
       RandomAccessIterator last);
void sort(RandomAccessIterator first,
       RandomAccessIterator last,StrictWeakOrdering comp);
```

trient l'intervalle `[first,last)` par ordre croissant suivant `<` ou `comp` respectivement.

La complexité moyenne est $O(Nlog(N))$ où N est le nombre d'éléments de la portée. Notons que ces tris ne s'appliquent qu'à des conteneurs à accès directs, mais d'autres conteneurs tels les `list` §10.10 ont des *méthodes* de tris qui ont de plus l'avantage de préserver les itérateurs. D'autre part les *Sorted Associative Container*s sont conservés triés.

```
void stable_sort(
        RandomAccessIterator first,RandomAccessIterator last);
void stable_sort(RandomAccessIterator first,
        RandomAccessIterator last,StrictWeakOrdering comp);
```

Ces deux fonctions sont similaires aux précédentes, sauf que les tris sont *stables*, c'est à dire que l'ordre des éléments équivalents est préservé.

La complexité est de $O(N(logN))$ quand l'algorithme dispose d'une mémoire auxiliaire suffisante et de $O(N(logN)^2)$ dans les autres cas.

```
bool is_sorted(ForwardIterator first,ForwardIterator last)
bool is_sorted(ForwardIterator first,ForwardIterator last,
        StrictWeakOrdering comp)
```

vérifie que l'intervalle `[first,last)` est trié par ordre croissant suivant `<` ou `comp`.

La complexité de cet algorithme est linéaire.

```
OutputIterator merge(InputIterator1 first1,InputIterator1 last1,
   InputIterator2 first2,InputIterator2 last2,
   OutputIterator result);
OutputIterator merge(InputIterator1 first1,InputIterator1 last1,
```

```
InputIterator2 first2,InputIterator2 last2,
OutputIterator result,StrictWeakOrdering comp);
```

fusionnent les deux intervalles triés de manière *stable*.

La complexité de `merge` est linéaire.

```
inline void inplace_merge(
  BidirectionalIterator first,BidirectionalIterator middle,
  BidirectionalIterator last);
inline void inplace_merge(
  BidirectionalIterator first,BidirectionalIterator middle,
  BidirectionalIterator last,StrictWeakOrdering comp);
```

fusionnent deux portées triées et contiguës `[first,middle)` et `[middle, last)` en une seule portée triée `[first, last)`.

La complexité est de $O(N)$ quand l'algorithme dispose d'une mémoire auxiliaire suffisante et de $O(N(logN))$ sinon.

Opérations ensemblistes.

Toutes les opérations ensemblistes de la bibliothèque STL s'appliquent exclusivement à des intervalles ordonnés. Il n'est pas demandé que les éléments soient tous distincts, un même élément peut apparaître plusieurs fois dans le même conteneur. Les opérations ensemblistes sont toutes de complexité linéaire.

```
OutputIterator set_union(
        InputIterator1 first1,InputIterator1 last1,
        InputIterator2 first2,InputIterator2 last2,
        OutputIterator result);
OutputIterator set_union(
        InputIterator1 first1,InputIterator1 last1,
        InputIterator2 first2,InputIterator2 last2,
        OutputIterator result,StrictWeakOrdering comp);
```

effectuent l'union de deux intervalles.

Les deux intervalles doivent être ordonnés ; le résultat est un intervalle ordonné qui contient chaque élément qui est contenu dans un des deux conteneurs. Si un élément est présent $n1$ fois dans le premier et $n2$ fois dans le second il sera présent $max(n1,n2)$ fois dans le résultat.

Dans le cas le plus courant où les deux intervalles sont des ensembles ordonnés, le résultat est alors également un ensemble ordonné.

La première version utilise `<` pour ordonner les éléments et la seconde `comp`.

```
OutputIterator
  set_intersection(InputIterator1 first1,InputIterator1 last1,
     InputIterator2 first2,InputIterator2 last2,
     OutputIterator result);
OutputIterator
  set_intersection(InputIterator1 first1,InputIterator1 last1,
     InputIterator2 first2,InputIterator2 last2,
     OutputIterator result,StrictWeakOrdering comp);
```

calculent l'intersection de deux ensembles.

Suivant le même principe que l'algorithme ci-dessus, un élément est présent $min(n1,n2)$ fois dans le résultat. Ainsi, si les deux conteneurs initiaux sont des ensembles ordonnés, le résultat est un conteneur ordonné qui est leur intersection.

```
bool includes(InputIterator1 first1,InputIterator1 last1,
              InputIterator2 first2,InputIterator2 last2);
bool includes(InputIterator1 first1,InputIterator1 last1,
              InputIterator2 first2,InputIterator2 last2,
              StrictWeakOrdering comp);
```

Rendent `true` si le premier intervalle est inclus dans le second.

Les intervalles `[first1,last1)` et `[first2,last2)` doivent être ordonnés. La valeur de retour de `includes` est vraie si tout élément de `[first1, last1)` est équivalent à un élément de `[first2,last2)`. L'équivalence est déterminée par `==` pour la première fonction, et `comp` pour la seconde.

```
OutputIterator set_difference(
  InputIterator1 first1,InputIterator1 last1,
  InputIterator2 first2,InputIterator2 last2,
  OutputIterator result);
  OutputIterator set_difference(
    InputIterator1 first1,InputIterator1 last1,
    InputIterator2 first2,InputIterator2 last2,
    OutputIterator result,StrictWeakOrdering comp);
```

effectuent une différence ensembliste *dans le cas ou les deux portées sont des ensembles*.

Dans le cas général où une valeur apparaît m fois dans `[first1,last1)` et n fois dans `[first2,last2)` elle apparaît $max(m-n,0)$ fois dans la portée de sortie.

C'est une opération stable : les éléments sont copiés de préférence à partir de la première portée, et leur ordre relatif est préservé.

```
set_symmetric_difference(
  InputIterator1 first1,InputIterator1 last1,
  InputIterator2 first2,InputIterator2 last2,
  OutputIterator result);
set_symmetric_difference(
  InputIterator1 first1,InputIterator1 last1,
  InputIterator2 first2,InputIterator2 last2,
  OutputIterator result,StrictWeakOrdering comp);
```

effectuent une différence ensembliste symétrique $(A\backslash B)\cup(B\backslash A)$ *dans le cas ou les deux portées sont des ensembles.*

Dans le cas général où une valeur apparaît m fois dans `[first1,last1)` et n fois dans `[first2,last2)` elle apparaît $|m-n|$ fois dans la portée de sortie.

10.3 Utilitaires généraux.

10.3.1 Opérateurs relationnels.

La STL définit dans l'en-tête `<utility>` des types patrons qui permettent de définir les opérateurs relationnels les uns en fonction des autres. Pour cela la bibliothèque se sert du mécanisme de la surcharge (cf. §9.4).

Si, par exemple, vous avez défini l'opérateur `==` pour un type *et pas* l'opérateur `!=` quand le compilateur trouve une référence à `operator !=`, il va générer automatiquement l'exemplaire patron de cet opérateur qui résout `x != x` comme `! (x == x)`.

Ainsi il suffit de définir `<` et `=`, pour générer `!=`, `<=`, `>=`, `>`.

L'instantiation des opérateurs relationnels est déclenchée dès le moment où l'en-tête `<utility>` est incluse. Comme les inclusions se propagent de manière transitive en suivant les `#include` le résultat d'une telle inclusion peut être d'erreurs difficiles à corriger. Il sera souvent préferable d' éviter d'inclure de manière systématique ces patrons avec l'en-tête `<utility>`.

Quand `<utility>` est inclus, pour éviter la génération pour un opérateur particulier il suffit de définir explicitement l'opérateur en question.

On consultera aussi la section 7.6.8 qui traite de la surcharge des opérateurs.

Les prototypes de ces fonctions patrons sont :

```
template <class T> bool operator!=(const T& x, const T& y);
template <class T> bool operator>(const T& x, const T& y);
template <class T> bool operator<=(const T& x, const T& y);
template <class T> bool operator>=(const T& x, const T& y);
```

10.3.2 La structure patron `pair`

`Pair<T1,T2>` est un doublet formé de deux valeurs de types hétérogènes. Il est utilisé par les fonctions pour retourner une paire de valeurs, comme le fait par exemple `mismatch` défini précédemment.

`first` désigne le premier objet de la paire et `first_type` son type. De même on utilise `second` et `second_type` pour le deuxième argument.

Nous pouvons construire les paires avec le constructeur standard par exemple :

```
  pair<float,bool>p1(0,false);
```

Mais nous pouvons aussi nous servir de la fonction patron `make_pair`

```
template <class T1,class T2>
  pair<T1,T2>make_pair(const T1& x,const T2& y);
```

qui nous permet une déclaration brève comme dans l'exemple suivant :

```
if ( mismatch (f1,l1,f2,l2)==make_pair(f1,f2)) ....
```

Quand l'égalité est définie pour les deux types d'arguments, elle est aussi définie pour les paires. Il en est de même pour `operator<`.

10.3.3 Les itérateurs de flot.

Les itérateurs de flots permettent un accès aux flots suivant les normes de la STL. Ils permettent de considérer les flots comme des conteneurs. Ce sont les itérateurs les plus *pauvres* de la bibliothèque, dans la mesure où leurs propriétés sont les plus réduites. Il est important en utilisant ces itérateurs de se souvenir des restrictions relatives à leur emploi. Elles proviennent du caractère irréversible de l'écriture ou de la lecture dans un flot.

Les itérateurs de sortie : `ostream_iterator`.

Un `ostream_iterator` est un itérateur de sortie qui écrit dans un flot de sortie. Il comporte les restrictions des `Output Iterator` en particulier le parcours unidirectionnel et l'impossibilité de lire la valeur déréférencée. (cf. §40). On utilise pour les créer les deux constructeurs suivants :

```
ostream_iterator(ostream& s)
ostream_iterator(ostream& s, const char* delim)
```

Avec le premier constructeur une affectation à la valeur pointée est équivalente à `s << t`, et avec le second constructeur elle est équivalente à `s <<t <<delim.` Exemple :

```
vector<int> V;
// ...
copy(V.begin(), V.end(), ostream_iterator<int>(cout, "\n"));
```

Les itérateurs d'entrée : `istream_iterator`.

Un `istream_iterator` est utilisé pour parcourir un flot en entrée. C'est un `InputIterator` qui comporte les restrictions de ce type (voir §40) ; en particulier après incrémentation l'ancienne valeur d'un itérateur d'entrée n'est plus valable.

Le programme 10.16 illustre l'utilisation des itérateurs d'entrée et de sortie.

Programme 10.16 Utilisation d'itérateurs sur des flots

```
#include<iostream>
#include<vector>
using std::cout; using std::endl;
using std::cin;

std::vector<int> v;
int main(){
   cout << "?" ;
   for ( std::istream_iterator<int>it(cin);
         it!=std::istream_iterator<int>(); ++it){
      v.push_back(*it);
   }
   cout << endl << "v:";
   copy(v.begin(), v.end(), std::ostream_iterator<int>(cout, " "));
   cout << endl;
   return 0;
}
```

Entrée

```
? 1 2 3 4 5 ^D
```

Résultat

```
v:1 2 3 4 5
```

10.4 Les objets-fonctions

10.4.1 Description.

Un objet-fonction est un objet qui peut être appelé comme une fonction : ce peut être une fonction, un pointeur sur une fonction, ou encore un membre d'une classe qui définit la méthode `operator()`.

Un exemple d'objet-fonction est donné par l'objet de la classe `less_mag` qui teste les valeurs absolues de deux `double`.

```
struct less_mag: public binary_function<double, double, bool>
{
bool operator()(double x, double y) {return fabs(x) < fabs(y);}
};
```

qui peut être utilisé comme suit :

```
vector<double> V;
  ...
sort(V.begin(), V.end(), less_mag());
```

`less_mag` est une classe qui a un seul objet, puisque c'est une classe sans données membres et avec le seul constructeur par défaut. Elle possède une méthode `operator()` et l'objet `less_mag()` qui est passé à la fonction `sort` est donc un objet-fonction.

Les classes qui définissent une méthode `operator()` et dont les objets sont donc des objets-fonctions sont appelées des *classes-fonctions*.

Pour les classes qui n'ont qu'un objet on confondra objet-fonction et classe-fonction.

`less_mag` a deux arguments de type `double` : c'est une fonction binaire. Pour aider à l'utilisation des classes-fonctions binaires le type de base `binary_function` est défini dans la STL par :

```
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

`binary_function` et `unary_function` ne fournissent que des `typedef` elles n'ajoutent rien à la classe-fonction, mais la rendent compatible avec l'interface d'utilisation de la STL. Les classes-fonctions qui respectent cet interface et exportent donc les types `first_argument_type`, `second_argument_type`

Programme 10.17 Classe pour écrire un tableau.

```
template <class E> class Ecrire {
  public:
    Ecrire(std::ostream& os):osM(os){}
    std::ostream & operator()(E e){
    return osM<< e.val() << ", ";
  }
   private:
    std::ostream& osM;
};

template <typename InputIterator>
void ecrit(InputIterator deb, InputIterator fin){
  Ecrire<std::iterator_traits<InputIterator>::value_type > f(cout);
  std::for_each(deb,fin,f);
  cout << endl;
}
```

et result_type sont dits *adaptables*, les classes de base binary_function et unary_function aident à définir des classes adaptables.

Parmi les binary_functions les plus utilisées figurent les Binary Predicates ou prédicats binaires, qui sont des (objets) fonctions binaires qui rendent un booléen.

Nous utiliserons aussi souvent des classes-fonctions unaires qui seront rendues adaptables en les dérivant de :

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};
```

Un exemple est la classe Additionneur du programme 10.11. Cette classe modifie son état à chaque appel ce qui lui permet de calculer une somme.

Une classe-fonction peut permettre de définir plusieurs objets-fonctions si son constructeur admet au moins un argument c'est le cas de la classe-fonction patron 10.17 qui est utilisée dans le programme 7.4

Un *prédicat* où Predicate est un objet-fonction qui rend un booléen.

Enfin les objets-fonctions sans argument, encore appelés *générateurs* seront utilisés pour initialiser les éléments des conteneurs, le plus souvent avec l'algorithme generate (voir §10.2.2)

10.4.2 Les objets-fonctions prédéfinis.

Des objets-fonctions *adaptables* correspondants à toutes les opérations usuelles sont prédéfinis dans la STL. Il s'agit des opérations arithmétiques (plus, minus, multiplies, divides, modulus, et negate), des comparaisons (equal_to, not_equal_to greater, less, greater_equal, et less_equal) et les opérations logiques (logical_and, logical_or, et logical_not).

Un exemple d'utilisation de multiplies figure dans le programme 10.18

Toutes ces classes-fonctions sont définies de manière similaire dans <functional>, comme par exemple :

```
template <class T>
struct greater : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const {
        return x > y;
     }
};
```

10.4.3 Les adaptateurs.

Les adaptateurs[4] sont des fonctions patrons qui prennent pour arguments un ou plusieurs objets fonctions *adaptables* et délivrent en résultat un autre objet-fonction.

Les adaptateurs correspondent à des fonctions d'ordre supérieur[5] qui prennent leur valeurs et leur résultats dans un espace fonctionnel. Cette possibilité existe depuis le début dans les langages applicatifs comme LISP, *Scheme* ou CAML mais il est très intéressant que les patrons en permettent un usage, certes limité, mais sans aucunement sacrifier l'efficacité d'un langage compilé à typage et liaison statiques.

Certains adaptateurs sont prédéfinis dans la bibliothèque et accessibles par l'en-tête <functional>.

L'adaptateur pointeur sur fonction : ptr_fun

Il transforme un pointeur sur une fonction unaire (respectivement binaire) en objet-fonction adaptable de type Pointer_to_unary_function (resp. Pointer_to_binary_function).

[4] il semble que le terme soit ambigu dans la littérature : on nomme parfois adaptateur la fonction d'ordre supérieur, et parfois la classe-fonction qui contient le résultat

[5] appelés *fonctionnelle* en mathématiques

Il est théoriquement possible d'utiliser directement ces classes adaptateurs, mais cela demande d'instancier directement les arguments patrons. Quand on utilise le patron de fonction ptr_fun, le mécanisme de résolution des surcharges du compilateur se charge de déterminer quels sont les types nécessaires.

Ainsi la fonction de bibliothèque fabs peut être utilisée comme argument effectif quand un objet-fonction unaire est attendu comme par exemple dans :

```
transform(first, last, first, fabs);
```

Mais quand un objet-fonction *adaptable* est demandé alors fabs ne convient pas et on devra utiliser ptr_fun(fabs).

L'adaptateur pointeur sur fonction membre : mem_fun

De manière identique à ptr_fun ci-dessus, mem_fun transforme un pointeur sur une méthode en objet fonction. Si f est une méthode sans argument d'une classe X, mem_fun(&X::f) est un objet-fonction qui prend en argument un pointeur X* et qui est équivalent à p->f(), mais peut être utilisé par les algorithmes standards.

Un exemple est donné §8.4.1

Pour une méthode f à un argument de type A d'une classe X, mem_fun(&X::f) est un objet-fonction qui prend deux arguments : un pointeur p de X* et une valeur a de type A et qui est équivalent à p->f(a), mais peut être utilisé par les algorithmes standards.

L'adaptateur pointeur sur fonction membre : mem_fun_ref

Comme mem_fun, mem_fun_ref transforme un pointeur sur une méthode en objet fonction, mais l'argument de cet objet fonction au lieu d'un pointeur sur un objet est une référence à l'objet.

Si f est une méthode sans argument (respectivement avec un argument de type A) d'une classe X, mem_fun_ref(&X::f) est un objet-fonction qui prend en argument une référence à un objet x de X (resp. et un argument a de type A) et qui est équivalent à x.f() (resp. x.f(a)), mais peut être utilisé par les algorithmes de la STL ainsi que les adaptateurs.

Le programme 10.21 donne un exemple d'utilisation de mem_fun_ref.

Les adaptateurs de négation : not1 et not2

not1 (respectivement not2) transforme un prédicat unaire (resp. binaire) adaptable en un autre prédicat unaire (resp. binaire) adaptable qui est la négation du premier.

Ce second prédicat est un membre de la classe unary_negate<Adaptable-Predicate> (resp. binary_negate<AdaptablePredicate> que nous n'aurons pas à utiliser directement.

Programme 10.18 Utilisation de `bind1st`

```
#include <vector>
#include <iostream>
#include <functional>
#include <numeric>

std::vector<int> v(10);

int main(){
   iota(v.begin(), v.end(), 0);
   std::transform(v.begin(), v.end(), v.begin(),
                  std::bind1st(std::multiplies<int>(),));
   std::copy(v.begin(), v.end(),
             std::ostream_iterator<int>(std::cout, " "));
}
```

Résultat

```
0 2 4 6 8 10 12 14 16 18
```

Les adaptateurs de liaison : `bind1st` et `bind2nd`

`bind1st` (respectivement `bind2nd`) prend pour arguments un objet-fonction binaire adaptable et une valeur du type de son premier (resp. second) argument et délivre l'objet-fonction *unaire* adaptable obtenu en *liant* le premier (resp. le deuxième) argument à la valeur.

Le programme 10.18 illustre l'utilisation de `bind1st` ; `multiplies<int>()` est l'objet-fonction binaire qui est prédéfini dans la bibliothèque STL. Il effectue la multiplication de deux entiers comme le fait `operator*(int,int)` mais il est de plus *adaptable*.

`bind1st(multiplies<int>(),2)` est l'objet-fonction *unaire* obtenu en liant le multiplicateur à 2 ; c'est donc simplement une multiplication par 2. La suite des entiers de 0 à 9 obtenue par `iota` est donc transformée en la suite des nombres pairs de 0 à 18 par `transform`.

Résultat :

```
nom Mois : août
```

Dans le programme 10.19 la fonction `strcmp` est d'abord transformée en objet-fonction adaptable par `ptr_fun`, puis on lie son second argument à la chaîne de caractères `"août"` pour obtenir avec l'expression `(bind2nd(ptr_fun(strcmp),"août"))` une comparaison avec la chaîne `"août"` du premier argument. Le résultat est toujours celui de `strcmp` c'est-à-dire -1 ou 1 quand

Programme 10.19 Utilisation de `bind2nd`

```
#include <iostream>
#include <functional>
const char * nomMois[]={"janvier", "février", "mars", "avril", "mai","juin",
     "juillet", "août", "septembre", "octobre", "novembre", "décembre"};

int main(){
   const char** m= std::find_if(nomMois,nomMois+12,
      std::not1(std::bind2nd(std::ptr_fun(strcmp),"août")));
   if (m < nomMois+12){
      std::cout <<"nom Mois: " << *m << ", num: "<< m-nomMois << std::endl;
   }else{
      std::cout <<"non trouvé"<< std::endl;
   }
   return 0;
}
```

Résultat

```
nom Mois: août, num: 7
```

les chaînes sont différentes et 0 quand elles sont égales. L'adaptateur `not1` nie cette expression pour en faire une comparaison standard.

Notre `find_if` va donc examiner toutes les chaînes et rendre un pointeur sur la première égale à `"août"`, ce que nous vérifions en faisant écrire la chaîne trouvée et sa position dans le tableau.

Adaptateurs de composition : `compose1` et `compose2`

`compose1` (respectivement `compose2`) prend en argument deux objets-fonctions unaires (resp. binaires) adaptables et délivre l'objet-fonction unaire qui constitue la composée de ces fonctions.

Le programme 10.20 illustre la composition de deux objets-fonctions simple, 10.21 la composition d'une fonction et d'une méthode de classe alors que 10.22 utilise une composition binaire pour chercher le premier élément compris entre 1 et 10 d'une liste.

Programme 10.20 Utilisation de `compose1`

```
#include <string>
#include <iostream>
#include <functional>
using std::cout; using std::endl;
int deuxfois(int i){return 2*i;}
int suiv(int i){return i+1;}
struct Incr: public std::unary_function<int,int>{
      Incr(){}
      int operator () (int i)const{return i+1;}
};
struct DeuxFois: public std::unary_function<int,int>{
      DeuxFois(){}
      int operator () (int i)const{return 2*i;}
};
int main(){
   Incr inc;
   cout << inc(3)<<endl;
   cout << DeuxFois()(4)<<endl;
   cout <<compose1(inc,DeuxFois())(3)<<endl;
   std::unary_compose<Incr,DeuxFois>f=compose1(inc,DeuxFois());
   cout<< f(1)<<", "<< f(2)<<", "<< f(3)<<", "<< f(4)<<", "<<endl;
   cout <<std::ptr_fun(deuxfois)(4)<<endl;
   cout <<compose1(std::ptr_fun(suiv),std::ptr_fun(deuxfois))(3)<<endl;
  return 0;
}
```

Résultat

```
4
8
7
3, 5, 7, 9,
8
7
```

Programme 10.21 Utilisation de `compose1` pour une méthode

```
#include <string>
#include <iostream>
#include <functional>
using std::cout; using std::endl;

int main(){
   std::string s("toto");
   cout << std::mem_fun_ref(&std::string::c_str)(s)<<endl;
   int l=std::compose1(std::ptr_fun(strlen),
                       std::mem_fun(&std::string::c_str)) (&s);
   cout << l << ", "<<s.length()<<endl;
  return 0;
}

```

Résultat

```
toto
4, 4
```

Programme 10.22 Utilisation de `compose2`

```
list<int> L;
...
list<int>::iterator in_range =
     find_if(L.begin(), L.end(),
             compose2(logical_and<bool>(),
                      bind2nd(greater_equal<int>(), 1),
                      bind2nd(less_equal<int>(), 10)));
```

10.5 Le conteneur map

map<Key,Data,Compare,Alloc>

Les map sont des conteneurs qui réalisent un tableau associatif. Ils associent un objet de type Key à un objet de type Data. Un conteneur de type map a pour éléments des paires (voir §10.3.2) de type pair<const Key,Data>. Deux éléments distincts doivent toujours avoir des clés distinctes, et si cela ne peut être réalisé on utilisera un multimap.

Contrairement aux vecteurs et listes, les itérateurs sur un tableau associatif ne sont pas invalidés par une insertion ou l'effacement d'un autre élément que l'élément pointé.

10.5.1 Paramètres patrons

Key : le type des clés.

Pour tous les conteneurs associatifs la clé d'un élément ne change jamais, de son insertion à sa suppression. Les maps sont de plus des conteneurs à clé unique ; deux éléments ne peuvent avoir la même clé.

Data : Le type des données.

L'affectation doit être définie sur ce type.

Compare : la fonction de comparaison des clés.

Elle a pour défaut less<Key> qui elle-même se définit par défaut comme réalisant l'opérateur <. Compare doit réaliser un ordre partiel strict. Notons que comme pour tout conteneur associatif les clés n'ont pas obligatoirement d'égalité définie ; deux clés sont considérées comme équivalentes si aucune n'est plus petite que l'autre.

Alloc : l'allocateur utilisé pour la gestion mémoire. Il a pour défaut alloc.

10.5.2 Membres

key_type *(Associative Container)* : le type de clé. types

data_type *(Pair Associative Container)* : le type de valeur associée à la clé.

value_type *(Pair Associative Container)* : le type d'objet stocké dans la map, i.e. pair<const key_type,data_type>

`key_compare` *(Sorted Associative Container)* : objet fonction servant à trier les clés.

`value_compare` *(Sorted Associative Container)* : objet fonction servant à trier comparer les valeurs.

`pointer` *(Container)* : pointeur sur `CharT`.

`reference` *(Container)* : référence à `CharT`

`const_reference` *(Container)* : référence constante à `CharT`.

`difference_type` *(Container)* : un type entier signé.

C'est le type d'une différence d'itérateurs.

`size_type` *(Container)* : type entier non signé.

C'est le type d'une distance positive entre itérateurs.

itérateurs `iterator` *(Container)*

Itérateur utilisé pour parcourir un tableau associatif. Les itérateurs sur des `map` ne permettent pas de modifier les objets du conteneur car `map::value_type` n'admet pas d'affectation (*même si* `map::data_type` *doit, lui, accepter l'affectation*). Autrement dit pour un itérateur `i`, `*i=p` n'est pas une expression valide, mais `(*i).second=v` est valide. Ce qui explique qu'un itérateur de `map` n'appartient ni à la catégorie des itérateurs *mutable*, ni à celle des itérateurs constants.

`const_iterator` *(Container)*

Itérateur constant utilisé pour parcourir un tableau associatif.

`reverse_iterator` *(Reversible Container)*

Itérateur utilisé pour parcourir un tableau associatif de la fin vers le début. De la même manière que le type `iterator` ci-dessus il n'est ni *mutable*, ni constant.

`const_reverse_iterator` *(Reversible Container)*

Itérateur constant utilisé pour parcourir un tableau associatif vers l'arrière.

iterator begin() *(Container)*
retourne un iterator qui pointe au début du tableau.

iterator end() *(Container)*
retourne un iterator qui pointe à la fin du tableau.

const_iterator begin() const *(Container)*
retourne un const_iterator qui pointe au début du tableau.

const_iterator end() const *(Container)*
retourne un const_iterator qui pointe au début du tableau.

reverse_iterator rbegin() *(Reversible Container)*
retourne un reverse_iterator pour commencer un parcours inverse du tableau.

reverse_iterator rend() *(Reversible Container)*
retourne un reverse_iterator pour finir un parcours inverse du tableau.

const_reverse_iterator rbegin() const *(Reversible Container)*
retourne un const_reverse_iterator pour commencer un parcours inverse du tableau.

const_reverse_iterator rend() const *(Reversible Container)*
retourne un const_reverse_iterator pour finir un parcours inverse du tableau.

size_type size() const *(Container)* accesseurs
retourne la taille du tableau associatif.

size_type max_size() const *(Container)*
retourne la plus grande taille possible pour le tableau.

bool empty() const *(Container)*
retourne true si la taille est 0.

key_compare key_comp() const *(Sorted Associative Container)*
retourne l'objet fonction utilisé pour comparer les clés.

value_compare value_comp() const *(Sorted Associative Container)*
retourne l'objet fonction utilisé pour comparer les valeurs.

constructeurs map() *(Container)*
crée un tableau associatif vide.

map(const key_compare& comp) *(Sorted Associative Container)*
crée un tableau associatif vide en utilisant comp pour comparer les clés.

template <class InputIterator>
map(InputIterator f,InputIterator l) *(Unique Sorted Associative Container)*
crée un tableau associatif avec une copie de la portée [f,l).

template <class InputIterator>
map(InputIterator f,InputIterator l,
const key_compare& comp) *(Unique Sorted Associative Container)*
crée un tableau associatif avec une copie de la portée [f,l) en utilisant comp pour comparer les clés.

map(const map& m) *(Container)*
construit une copie de m.

affectation map& operator=(const map&) *(Container)*
opérateur d'affectation.

void swap(map&) *(Container)*
échange deux tableaux associatifs.

insertion pair<iterator,bool>insert(const value_type& x) *(Unique Sorted Associative Container)*
insère la paire x dans le tableau associatif *si la clé n'est pas déjà présente*.

Le premier élément de la paire retournée est un pointeur sur l'élément inséré ou l'élément de même clé déjà présent. Le second élément est true si l'élément a pu être inséré et false si un élément de même clé était déjà présent.

pair<iterator,bool>insert(const iterator pos,value_type& x)
(Unique Sorted Associative Container)
insère x dans le tableau associatif si la clé n'est pas déjà présente.

La position utilisée pour commencer la recherche de la clé est `pos`. La valeur de retour est un itérateur sur l'élément de même clé présent dans le tableau après l'opération.

`template <class InputIterator>`
`void insert(InputIterator first,InputIterator last)` *(Unique Sorted Associative Container)*
insère la portée `[first,last)` dans le tableau.

Cette opération est équivalente à une suite de `insert(*p)` où `p` parcourt tous les itérateurs de la portée `[first,last)`

`data_type& operator[](const key_type& k)` *(map)*
retourne une référence à l'objet de clé `k`.

Si un tel objet n'existe pas *il est créé* avec la valeur par défaut `data_type()`. Ce fonctionnement est caractéristique des conteneurs associatifs de la STL, et les différencie des conteneurs tels que les vecteurs. Pour un vecteur c'est une erreur d'accéder à un élément d'index inexistant, pour une `map` il est correcte de créer un objet en affectant une valeur à une clé inexistante.

Puisque `operator[]` peut créer un élément il n' existe pas de version constante de cette méthode.

`iterator erase(iterator p)` *(Associative Container)* suppression
efface l'élément pointé par `p`.

`size_type erase(const key_type& k)` *(Associative Container)*
efface les éléments de clé `k`, et retourne le nombre d'éléments effacés. Ici puisque la clé est unique 0 ou 1 élément sont effacés.

`iterator erase(iterator first,iterator last)` *(Associative Container)*
efface l'intervalle `[first,last)`.

`void clear()` *(Associative Container)*
efface tous les éléments du tableau.

`iterator find(const key_type& k)` *(Associative Container)* recherche
recherche un élément de clé `k`, retourne un itérateur sur cet élément ou end() si il n'existe pas d'élément de clé `k`.

`const_iterator find(const key_type& k) const` *(Associative Container)*
recherche un élément de clé `k`, retourne un itérateur sur cet élément ou end() si il n'existe pas d'élément de clé `k`.

`size_type count(const key_type& k)` *(Associative Container)*
retourne le nombre d'élément dont la clé est `k`.

Pour une `map` la clé est unique et l'algorithme retourne 0 ou 1.

`iterator lower_bound(const key_type& k)` *(Sorted Associative Container)*
renvoie un itérateur sur le premier élément dont la clé n'est pas plus petite que `k` ou `end()` si un tel élément n'existe pas.

Comme il s'agit ici d'un conteneur à clé *unique* si le conteneur contient un élément de clé égale à `k`, un itérateur sur cet élément est retourné.

`const_iterator lower_bound(const key_type& k) const` *(Sorted Associative Container)*
trouve le premier élément dont la clé n'est pas plus petite que `k`.

`iterator upper_bound(const key_type& k)` *(Sorted Associative Container)*
trouve le premier élément dont la clé est plus grande que `k` ou `end()` si un tel élément n'existe pas.

Comme il s'agit ici d'un conteneur à clé *unique* si le conteneur contient un élément de clé égale à `k`, un itérateur sur l'élément suivant est retourné.

`const_iterator upper_bound(const key_type& k) const` *(Sorted Associative Container)*
trouve le premier élément dont la clé est plus grande que `k`.

`pair<iterator,iterator>equal_range(const key_type& k)` *(Sorted Associative Container)*
retourne une portée dont tous les éléments ont pour clé `k`.

Cette portée est `[a.lower_bound(k),a.upper_bound(k))`. Ici, puisque la clé est unique cette portée, a au maximum un élément.

10.5.3 Fonctions non membres

`bool operator==(const map&,const map&)` *(Forward Container)*
teste l'égalité de deux *Forward Container*s en vérifiant qu'ils sont de même taille et que les éléments sont égaux deux à deux.

`bool operator<(const map&,const map&)` *(Forward Container)*

Comparaison lexicographique des conteneurs, équivalent à `lexicographical_compare` (voir 10.2.1).

10.6 Le conteneur `multimap`

`multimap<Key,Data,Compare,Alloc>`

Quand nous n'exigeons plus que la clé soit unique, nous obtenons les `multimap` qui sont encore des *Sorted associative Container*s mais plus des *Unique Sorted associative Container*s.

Cela implique que l'ajout d'un élément dans un `multimap` ne remplace pas un élément de même clé, mais vient s'y ajouter. Dans un `multimap` les éléments de même clé sont stockés de manière contiguë.

Les membres de `multimap` forment un sous-ensemble des membres de `map`, les opérations spécifiques aux conteneurs à clé unique (*Unique Sorted associative Container*) disparaissant du protocole.

10.7 Le conteneur `hash_map`

`hash_map<Key,Data,HashFcn,EqualKey,Alloc>`

Les `hash_map`s sont des conteneurs associatifs qui associent des clés de type `Key` à des valeurs de type `Data`. Comme les `map`s ce sont des conteneurs de type *Unique Associative Container*, autrement dit la clé identifie de manière unique l'élément qui est de type `pair<const Key,Data>`.

La principale différence avec les `map`s est que les éléments ne sont pas triés suivant la clé, mais qu'on y accède par adressage dispersé ; cela permet d'atteindre une complexité moyenne en temps constant. Les *Hashed Associative Container* sont donc beaucoup plus efficaces que les *Sorted Associative Container* quand l'ordre des éléments est sans importance.

10.7.1 Paramètres patrons

`Key` : le type des clés.

`Data` : le type des données.

L'affectation doit être définie sur ce type.

`HashFcn` : la fonction utilisée pour l'adressage dispersé.

La valeur par défaut de `HashFcn` est `hash<Key>`. `hash<Key>` est un objet fonction défini par défaut pour les types `char*`, `const char*`, `crope`, `wrope`, et les types prédéfinis entiers. Si un autre type de clé est choisi nous devons donner la fonction de hachage ou spécialiser `hash<T>`.

`EqualKey` : la fonction d'égalité des clés.

Il est requis que `EqualKey` soit un prédicat binaire qui réalise une relation d'équivalence. Il a pour défaut `equal_to<Key>` qui réalise l'opérateur `==` pour le type `Key`.

`Alloc` l'allocateur utilisé pour la gestion mémoire.

Il a pour défaut l'allocateur standard `alloc`.

10.7.2 Membres

Les fonctions membres sont essentiellement identiques à celle des `map` à l'exception de celles spécifiques aux *(Sorted Associative Container)* qui sont remplacées par les membres des *Hashed Associative Container*. suivants :

`size_type bucket_count() const` *(Hashed Associative Container)*

`void resize(size_type n)` *(Hashed Associative Container)*

`hasher hash_funct() const` *(Hashed Associative Container)*

La fonction d'adressage utilisée par le `hash_map`.

`key_equal key_eq() const` *(Hashed Associative Container)*

Le prédicat d'égalité sur les clés.

`hash_map(size_type n)` *(Hashed Associative Container)*

crée un tableau d'adressage dispersé vide de taille initiale `n` avec la fonction d'adressage `h`.

`hash_map(size_type n,const hasher& h)` *(Hashed Associative Container)*

crée un tableau à adressage dispersé vide de taille initiale `n` avec la fonction d'adressage `h`.

`hash_map(size_type n,const hasher& h,const key_equal& k)` *(Hashed Associative Container)*

crée un tableau à adressage dispersé vide de taille initiale n avec la fonction d'adressage h et la fonction d'égalité des clés k.

hash_map(const hash_map&) *(Container)*
constructeur de copie.

data_type& operator[](const key_type& k) *(map)*
retourne une référence à l'objet de clé k, si un tel objet n'existe pas il est créé. Cet opérateur est similaire à celui de map [6]

10.8 Le conteneur hash_multimap

hash_multimap<Key,Data,HashFcn,EqualKey,Alloc>

Quand nous n'exigeons pas que la clé d'un hash_map soit unique, nous obtenons les hash_multimap qui sont encore des *Hashed associative Container*s mais plus des *Unique Hashed associative Container*s.

Cela implique que l'ajout d'un élément dans un hash_multimap ne remplace pas un élément de même clé, mais vient s'y ajouter. Dans un hash_multimap les éléments de même clé sont stockés de manière contiguë.

Les membres de hash_multimap forment un sous-ensemble des membres de hash_map, les opérations spécifiques aux conteneurs à clé unique (*Unique Hashed associative Container*) disparaissant du protocole.

10.9 Le conteneur deque

deque<T,Alloc>

Les deque sont des conteneurs très proches des vector. De la même manière ils permettent un accès direct aux éléments, des insertions et suppressions en temps constant à la fin du conteneur, et des insertions et suppressions en temps linéaire en milieu de conteneur.

Contrairement aux vectors les deques permettent des insertions et suppressions en temps constant au début du conteneur, mais il ne possèdent pas les membres capacity() et reserve() ni la garantie de validité des itérateurs associée à la possibilité de réservation de place *à la demande*. Les deque sont déclarés dans l'en-tête <deque>.

[6]Ce n'est cependant pas une opération des *Unique Associative Container*s car elle n'est pas supportée par set ou hash_set

10.9.1 Nouveaux membres.

En plus des membres déjà vu pour les vecteurs (10.1.1, 10.1.2, 10.1.1, 10.1.3, 10.1.4) qui sont des membres des protocoles *Random access container* et *Front insertion sequence*, nous avons de plus les membres de *Front Insertion Sequence* :

`void push_front(const T&)` *(Front Insertion Sequence)*
: Insère un élément au début de la suite.

`void pop_front()` *(Front Insertion Sequence)*
: Supprime le premier élément.

10.10 Le conteneur `list`.

`list<T,Alloc>`

Le conteneur `list` implémente des listes doublement chaînées. Il supporte des insertions et suppressions en temps constant à toutes les positions. De plus contrairement aux `vector` et `deque` les itérateurs ne sont pas invalidés par une insertion ou la suppression d'un autre élément que l'élément pointé. Cependant les relations de prédécesseur, et de successeur ne sont pas conservées par les insertions, suppressions ; autrement dit deux itérateurs consécutifs pointeront encore vers les mêmes éléments après une insertion (ou suppression d'un autre élément que les deux éléments pointés), mais ils ne seront pas obligatoirement contigus.

10.10.1 Méthodes spécifiques des listes

Les méthodes de la classe `list` sont les méthodes des *Reversible Container*, *Back Insertion Sequence* déjà vues pour les `vector`s (§10.1) et *Front Insertion Sequence* décrites pour les `deque`s (§10.9) auxquelles s'ajoutent les méthodes spécifiques suivantes :

`size_type size() const` *(Container)*
: Bien que cette méthode appartienne au protocole de *Container* elle est répétée ici car contrairement aux `vector`s et `deque`s, où elle est en temps constant, elle est en temps linéaire pour les listes. Il est donc beaucoup plus efficace pour une liste de tester `l.empty()` que `l.size()==0`.

`void splice(iterator position,list<T,Alloc>& x)` *(list)*
: transfère tout les éléments de `x` dans la liste avant `position`.

  Les éléments sont supprimés de `x`, cependant tous les itérateurs, ceux sur les éléments de la liste initiale, comme ceux sur les éléments de `x` restent valides. Il est requis que `x` soit distinct de la liste courante.

Cette fonction est en temps constant.

`void splice(iterator position,list<T,Alloc>& x,iterator i)` *(list)*

L'itérateur `i` doit pointer sur un élément de la liste `x` (qui *ici* peut être identique à `*this`), l'élément pointé est retiré de `x` et inséré avant `position`. L'opération est sans effet si `position==i` ou `position==++i`.

Cette fonction est en temps constant.

`void splice(iterator position,list<T,Alloc>& x,`
`iterator first,iterator last)` *(list)*

La portée `[first,last)` est transférée avant `position` tous les itérateurs sont conservés même ceux sur `x`. L'itérateur `position` ne doit pas pointer sur un élément de la portée déplacée. La fonction est en temps constant.

`void remove(const T& val)` *(list)*

supprime tous les éléments égaux à `val`, les itérateurs sur les autres éléments sont conservés.

Cette fonction est de temps linéaire et effectue exactement `size()` comparaisons.

`template<class Predicate>void remove_if(Predicate p)` *(list)*

supprime tous les éléments qui vérifient le prédicat `p`. Les itérateurs sur les autres éléments sont conservés.

La fonction est de temps linéaire.

`void unique()` *(list)*

ne garde que le premier élément de tout groupe d'éléments consécutifs égaux.

L'ordre des éléments conservés et les itérateurs sur ces éléments sont inchangés.

Cette fonction effectue `size()-1` comparaisons.

`void unique(BinaryPredicate p)` *(list)*

ne garde que le premier élément de tout groupe d'éléments consécutifs équivalents par le prédicat binaire `p`.

L'ordre des éléments conservés et les itérateurs sur ces éléments sont inchangés.

Cette fonction effectue `size()-1` comparaisons.

`void merge(list<T,Alloc>& x)` *(list)*

Les deux listes `*this` et `x` doivent être distinctes, et ordonnées par `<` ; les éléments de `x` sont transférés dans `*this` et insérés suivant l'ordre déterminé par `<`. La fusion est stable, c'est à dire que l'ordre relatif des éléments est conservé. Les itérateurs sur ces éléments sont préservés.

Cette fonction est linéaire et effectue au plus `size()+x.size()-1` comparaisons.

`void merge(list<T,Alloc>& x,BinaryPredicate Comp)` *(list)*

Cette méthode est similaire à la précédente, mais la comparaison s'effectue avec le comparateur `Comp` qui doit définir un ordre partiel strict (*LessThanComparable*).

`void reverse()` *(list)*

renverse l'ordre des éléments de la liste.

Les itérateurs sur les éléments de la liste sont préservés.

Cette fonction est en temps linéaire.

`void sort()` *(list)*

trie la liste suivant l'opérateur `<`.

Les itérateurs sur les éléments de la liste sont préservés.

C'est un tri stable en temps $O(Nlog(N))$.

`void sort(BinaryPredicate comp)` *(list)*

trie la liste suivant `Comp`

Le comparateur `Comp` doit définir un ordre partiel strict (*Less Than Comparable*). Les itérateurs sur les éléments de la liste sont préservés.

C'est un tri stable en temps $O(Nlog(N))$.

10.11 Le conteneur `set`.

`set<Key,Compare,Alloc>`

Les `set`s représentent des ensembles d'objet de type `Key`. Ce sont des *Sorted Associative Container* qui ont le même type pour leurs valeurs et leurs clés.

Un élément est présent au plus une fois, ce sont donc des *Unique Associative Container*.

Les `set` sont particulièrement adaptés aux opérations ensemblistes (parrefsec :op-ensembles) que procurent les algorithmes : `set_union`,

Programme 10.23 La soupe STL : Utilisation des sets.

```
#include <set>
#include <string>
#include <iostream>
typedef std::set<std::string> Legumes;
std::ostream& operator << (std::ostream& os,
                           const Legumes& legumes){
  copy(legumes.begin(), legumes.end(),
       std::ostream_iterator<std::string>(os, ", "));
  return os;
}
int main(){
   using std::cout; using std::endl; using std::string;
   Legumes soupe;
   Legumes reserve;
   reserve.insert("pommes de terres");
   reserve.insert("carottes");
   reserve.insert("noix");
   reserve.insert("riz");
   reserve.insert("orge");
   cout << "Réserve: "<<reserve<<endl;
   soupe.insert("carottes");
   soupe.insert("champignons");
   soupe.insert("orge");
   cout << "Soupe: "<<soupe<<endl;
   cout << "Ingrédients de la soupe en réserve (intersection):";
   cout << endl;
   set_intersection(reserve.begin(), reserve.end(),
             soupe.begin(), soupe.end(),
             std::ostream_iterator<string>(cout,", "));
   cout << endl<<"Courses (différence): ";
      set_difference( soupe.begin(), soupe.end(),
                      reserve.begin(), reserve.end(),
                      std::ostream_iterator<string>(cout,", "));
      cout << endl<<"Nouvelle réserve (union):"<< endl;
      set_union( soupe.begin(), soupe.end(),
                      reserve.begin(), reserve.end(),
                      std::ostream_iterator<string>(cout,", "));
      cout << endl
}
```

Programme 10.24 La soupe STL : résultat.

```
Réserve: carottes, noix, orge, pommes de terres, riz,
Soupe: carottes, champignons, orge,
Ingrédients de la soupe en réserve (intersection):
carottes, orge,
Courses (différence): champignons,
Nouvelle reserve (union):
carottes, champignons, noix, orge, pommes de terres, riz,
```

`set_intersection`, `set_difference`, `set_symmetric_difference`. En effet ce conteneur est trié et les opérations ensemblistes demandes des conteneurs triés et préservent l'ordre.

Les conteneurs `sets` sont une forme simplifiée des `maps` dont ils confondent clé et valeur.Comme les itérateurs de `map` les itérateurs des `set` ne sont pas invalidés par une insertion ou l'effacement d'un autre élément que l'élément pointé.

Un exemple d'utilisation de `set` est donné dans le programme 10.23 et 10.24
.

10.11.1 Paramètres patrons

`Key` : le type des clés.

c'est aussi le type de valeur. Il est aussi défini comme `set::key_type` et `set::value_type`

Comme la valeur d'un élément est aussi sa clé cette valeur ne peut jamais être modifiée, mais les objets peuvent être détruits et insérés.

`Compare` : la fonction de comparaison des clés.

Elle a pour défaut `less<Key>` qui elle-même se définit par défaut comme réalisant l'opérateur `<`. `Compare` doit réaliser un ordre partiel strict.

Elle est aussi définie comme `set::key_compare` et `set::value_compare`

`Alloc` : l'allocateur utilisé pour la gestion mémoire. Il a pour défaut `alloc`.

10.11.2 Membres

Tous les membres de la classe `set` sont des membres des `Unique Sorted Associative Container` et sont donc décrits pour la classe `map` §10.5.2 page 257.

10.12 Le conteneur multiset

multiset<Key,Data,Compare,Alloc>

Les multisets sont aux multimaps ce que les sets sont aux maps, des conteneurs associatifs ordonnés simples dont la clé est identique à la valeur.

Comme les sets les multiset sont adaptés à une utilisation efficace des opérations ensemblistes. La seule différence avec les sets est qu'un élément peut être présent en plusieurs exemplaires dans un multiset.

Les membres de multiset forment un sous-ensemble des membres de set, les opérations spécifiques aux conteneurs à clé unique (*Unique Sorted associative Container*) disparaissent du protocole.

Chapitre 11

La classe string

Ce que nous désignons comme classe string ou basic_string est une classe patron dont le nom complet est : basic_string<charT,traits,Alloc>

La classe basic_string fait partie des spécifications de la librairie standard (1985) et les bibliothèques usuelles proposent de l'implémenter avec un partage des données contrôlé par comptage des références.

Les implémentations disponibles de cette classe ont été longtemps ignorées par la STL et *Silicon Graphic (*SGI*)* qui en centralise le développement.

Les concepteurs de la STL reprochent aux implémentations des basic_string leur faible robustesse, en particulier en regard des processus légers (*threads*).

Depuis 1997 des versions de basic_string sont intégrées à la STL : elles satisfont les spécifications de la librairie standard en élargissant certaines restrictions. Surtout elles ont intégré les basic_string à la STL ; en en faisant des modèles des *Conteneurs à accès directs* et des *Suites*, ce qui permet de les manipuler de manière unifiée comme les autres conteneurs de la même sorte.

L'interface des basic_string est ainsi un peu lourd car il superpose les primitives des Sequences à celles de la librairie standard. Cependant la compatibilité aux normes est au prix de ces redondances.

Nous décrivons dans ce chapitre l'implémentation de la STL, notons que' ce n'est pas la seule version disponible, même dans les distributions récentes, si vous n'avez encore que les versions de la StdLib vous ne pourrez pas utiliser les primitives de Sequences, et vous devrez prendre garde a respecter de manière stricte les spécifications des méthodes c_str et data.

Les basic_string sont des des *Suites* (Sequence) comme les vector, deque, list, slist et des *Conteneurs à accès directs* (Random Access Container) comme les vector et deque. Ils ressemblent beaucoup aux vector<charT> mais ils permettent un interface aisé avec les chaînes terminées par un caractère nul, employées par les fonctions de **C**, et des entrées-sorties facili-

tées. En fait leur implémentation est complètement distincte des vector<charT> car ils stockent toujours un caractère nul à la fin de leur représentation pour permettre une réalisation rapide de c_str et aussi parce que les spécifications de la librairie standard diffèrent de celle des vector<charT> pour la copie des éléments [1].

11.1 Description

La classe basic_string est utilisée pour traiter les chaînes de caractères. Elle gère automatiquement les allocations et désallocations de mémoire. Les fonctions membres peuvent lancer des exceptions length_error et out_of_range. L'en-tête correspondant est <string> que l'on ne confondra pas avec l'en-tête **C** <string.h>

La classe basic_string est paramétrée par le type des caractères, mais usuellement on utilisera pas directement les basic_string mais les classes non paramétrées string et wstring qui sont définies respectivement comme basic_string<char> et basic_string<wchar_t>.

La classe basic_string contient les opérations des Sequences de la *STL* ainsi que des opérations supplémentaires qui sont des opérations usuelles sur les chaînes de caractères telles que les recherches et concaténations.

Comme les autres clases de conteneurs il est possible de donner la position dans une basic_string en utilisant un itérateur. Mais de nombreuses fonctions membres utilisent une valeur pos de type size_type pour représenter la position begin() + pos, ou deux valeurs, pos et n,pour représenter l'intervalle [begin() + pos,begin() + pos + n).

Bien que les normes ne précisent pas les performances des basic_string l'implémentation de la STL a des caractéristiques semblables à celle de vector. L'accès à un caractère est *O(1)*, alors que la copie et la concaténation sont linéaire c'est-à-dire *O(N)*. Notons qu'en revanche la classe rope a des itérateurs plus encombrants mais des opérations logarithmiques, ce qui la rend plus adaptée aux chaînes de grande taille.

Les normes de la bibliothèque standard donnent aux itérateurs des basic_string une validité plus restreinte que les normes de la STL. Les itérateurs sont bien sûr invalidés par swap, reserve, insert, et erase ; ainsi que par les fonctions équivalentes à insert et erase, telles que clear, resize, append, et replace. Mais, de plus, tout appel à une fonction membre non const invalide aussi les itérateurs. Cela est vrai même pour les versions non const de begin() ou operator[].

---

[1] la librairie standard impose l'utilisation de char_traits<>::assign, char_traits<>::copy, et char_traits<>::move

Programme 11.1 Utilisation de la classe `std : :string`.

»

```
#include<iostream>
#include<string>
int main() {
   using std::cout; using std::endl; using std::string;
  string s(10u, ' ');            // Une chaîne de 10 espaces

  const char* A = "c'est un test";
  s += A;
  cout << "string s -> \\n" << (s + '\n');
  cout << "Comme tableau terminé par un car. nul: \n" << s.c_str() << endl;
  cout << "quinzième caractère " << s[14] << endl;
  //les deux ordres suivants ne sont valides qu'avec la STL
  std::reverse(s.begin(), s.end());
  s.push_back('\n');
  cout << s;
  return 0;
}
```

Résultat

```
s =           c'est un test
Comme tableau terminé par un car. nul:
          c'est un test
quinzième caractère t
tset nu tse'c
```

Ces limites d'utilisation des itérateurs sont destinées à donner aux implémenteurs plus de liberté pour la gestion mémoire, cependant dans l'implémentation de la STL `begin()`, `end()`, `rbegin()`, `rend()`, `operator[]`, `c_str()`, et `data()` n'invalident pas les itérateurs. Dans cette implémentation, seules les fonctions qui changent le contenu de la chaîne invalident les itérateurs.

Deux exemples de programmes sont donnés figures 11.1 et 11.2. Le premier exemple montre un exemple simple de concaténation, le second exemple montre les différences entre le `string` et les tableaux terminés par zéro de `C`.

11.2 Paramètres patrons

– `charT` Le type de caractère que contient la chaîne.

Programme 11.2 `string` versus tableau terminé par zéro.

»

```
#include<iostream>
#include<cassert>
#include<cstring>
#include<string>

int main(int argc, char * argv[]){
   using std::cout; using std::endl; using std::string;
   char t[]={'u','n','\0','d','e','u','x','\0','t','r','o','i','s','\0'};
   cout <<"t=\""<< t<<"\""<<endl; // écrit t="un"
   assert(strcmp(t,"un")==0);
   assert(strcmp(t,"un\0deux\0trois")==0);
   string s (t,t+sizeof(t));
   cout <<"s=\""<< s<<"\""<<endl; //écrit s="un^@deux^@trois^@"
   assert(s==string("un")+'\0'+"deux"+'\0'+"trois"+'\0');
   assert (s!="un");
   assert (s!=string("un"));
   assert (s!="un\0deux\0trois");
   assert (s!="un\0deux\0trois\0");
   assert (s!=s.c_str());
   assert (s!=string(s.c_str()));
   assert (s ==string(s.data(),s.size()));
   assert (strcmp(s.c_str(),"un")==0);
   assert (strcmp(s.c_str(), "un\0sept")==0);
   assert (strcmp(s.c_str(), string("un\0sept",7).c_str())==0);
   assert (strcmp(s.c_str(),(string("un")+'\0'+"sept").c_str())==0);
   assert (s != (string("un")+'0'+"sept"));
   assert (memcmp(s.data(),"un\0deux\0trois\0",s.size())==0);
   cout << "fin des assertions"<<endl;
   return 0;
}
```

Résultat

```
t="un"
s="undeuxtrois"
fin des assertions
```

- traits Le type char_traits qui encapsule les opérations de base ; par défaut : char_traits<charT>
- Alloc L'allocateur utilisé pour la gestion mémoire interne ; par défaut : alloc

Ces types doivent vérifier les spécifications des (Random Access Container) (*Conteneurs à accès directs*) et des Sequence et de plus vérifier :

- charT is a POD («plain 'ol data»).
- traits a pour type de valeur charT

11.3 Membres

value_type *(Container)* types

Le type d'objet CharT, stocké dans la chaîne.

pointer *(Container)*

Pointeur sur CharT.

reference *(Container)*

Référence à CharT

const_reference *(Container)*

Référence constante à CharT.

size_type *(Container)*

Un type entier non signé.

difference_type *(Container)*

Un type entier signé, il est utilisé pour les positions relatives dans une chaîne.

static const size_type npos *(basic_string)*

La plus grande valeur du type size_type. C'est-à-dire size_type(-1). C'est la plus grande taille possible d'une chaîne.

iterator *(Container)* itérateurs

Itérateur utilisé pour parcourir une chaîne. basic_string fournit des itérateurs à accès directs.

const_iterator*(Container)*

Itérateur constant utilisé pour parcourir une chaîne.

reverse_iterator *(Reversible Container)*

Itérateur utilisé pour parcourir une chaîne vers l'arrière.

`const_reverse_iterator` *(Reversible Container)*
: Itérateur constant utilisé pour parcourir une chaîne vers l'arrière.

`iterator begin()` *(Container)*
: Retourne un `iterator` qui pointe au début de la chaîne.

`iterator end()` *(Container)*
: Retourne un `iterator` qui pointe à la fin de la chaîne.

`const_iterator begin() const` *(Container)*
: Retourne un `const_iterator` qui pointe au début de la chaîne.

`const_iterator end() const` *(Container)*
: Retourne un `const_iterator` qui pointe au début de la chaîne.

`reverse_iterator rbegin()` *(Reversible Container)*
: Retourne un `reverse_iterator` qui pointe au début de la chaîne inverse.

`reverse_iterator rend()` *(Reversible Container)*
: Retourne un `reverse_iterator` qui pointe à la fin de la chaîne inverse.

`const_reverse_iterator rbegin() const` *(Reversible Container)*
: Retourne un `const_reverse_iterator` qui pointe au début de la chaîne inverse.

`const_reverse_iterator rend() const` *(Reversible Container)*
: Retourne un `const_reverse_iterator` qui pointe à la fin de la chaîne inverse.

accesseurs

`size_type size() const` *(Container)*
: Retourne la taille de la chaîne.
: On a toujours `0<= size()` et `size() <= capacity()`

`size_type length() const` *(basic_string)*
: Synonyme pour `size()`.

`size_type max_size() const` *(Container)*
: Retourne la taille maximale de la chaîne.

`size_type capacity() const` *(basic_string)*
: Le nombre d'éléments pour lequel la mémoire est allouée, c'est-à-dire la taille à partir de laquelle la chaîne devra être réallouée cette quantité est toujours supérieure à `size()` et inférieure à `max_size()`.

`bool empty() const` *(Container)*
: `true` si la taille est `0`.
: Équivalent à `size()==0` (mais probablement plus rapide) .

`reference operator[](size_type n)` *(Random Access Container)*

Retourne le `n` ème caractère.

`const_reference operator[](size_type n) const` *(Random Access Container)*

Retourne le `n` ème caractère.

`const charT* c_str() const` *(basic_string)* conversion

Retourne un tableau de caractères terminé par zéro représentant le contenu de la chaîne. Pour toute chaîne `s.c_str()[s.size()]` est un caractère nul.

Remarquons que ce n'est pas toujours le premier caractère nul, la chaîne peut comprendre des caractères nuls.

`const charT* data() const` *(basic_string)*

Retourne un tableau de caractères (pas obligatoirement terminé par zéro) représentant le contenu de la chaîne.

`data()` peut être identique à `c_str()` mais ce n'est pas obligatoire. Ses premiers `size()` caractères sont identiques à ceux de `*this`. `data()` ne retourne jamais un pointeur nul, même si `size()` vaut zéro.

`basic_string()` *(Container)* construction

Crée une chaîne vide.

```
basic_string(const basic_string& s,size_type pos = 0,
             size_type n = npos)
```
(basic_string)

Constructeur de copie qui copie `n` caractères de `s` à partir de `pos`, la copie s'arrête à la fin de `s`, si elle survient avant la position `pos + n`. Avec les valeurs par défaut c'est un constructeur de copie standard.

Ce constructeur lance l'exception `out_of_range` si `pos >s.size()`.

`basic_string(const charT*)` *(basic_string)*

Construit une `basic_string` avec un tableau de caractères terminé par zéro.

Équivalent à `basic_string(s,s + traits::length(s))`.

`basic_string(const charT* s,size_type n)` *(basic_string)*

Construit une `basic_string` avec un tableau de caractères et une longueur.

Équivalent à `basic_string(s,s + n)`.

`basic_string(size_type n,charT c)` *(Sequence)*

Crée une chaîne avec `n` copies de `c`.

pré-condition : `0<= n`.

poscondition : `size()=n` et chaque élément est une copie de `c`.

`template <class InputIterator>`
`basic_string(InputIterator first,InputIterator last)` *(Sequence)*

Crée une chaîne avec un intervalle.

pré-condition : `first` et `last` sont des `InputIterator` dont le type de valeur est convertible en `charT` ; et `[first,last)` est un intervalle valide.

post-condition : `size()` est égal à la distance entre `first` et `last` et chaque élément de la chaîne est une copie de l'élément correspondant de la chaîne.

`~basic_string()` *(Container)*

Le destructeur.

affectation `basic_string& operator=(const basic_string&)` *(Container)*

L'opérateur d'affectation.

Après affectation `size()==s.size()` et chaque élément est une copie de l'élément correspondant de `s`

`basic_string& operator=(const charT* s)` *(basic_string)*

Affectation d'un tableau de caractères terminé par zéro à une chaîne.

Équivalent à `operator=(basic_string(s))`.

`basic_string& operator=(charT c)` *(basic_string)*

Affecte un caractère unique à une chaîne.

`void reserve(size_t) n` *(basic_string)*

Demande que la capacité soit changée pour une grandeur supérieure à `n`. La capacité peut être diminuée avec `reserve` mais elle ne devient jamais inférieure à `size()`.

`reserve()` lance l'exception `length_error` si `n >max_size()`.

`void swap(basic_string&)` *(Container)*

Échange le contenu de deux chaînes.

insertion `iterator insert(iterator pos,const T& x)` *(Sequence)*

Insère `x` avant `pos`.

pré-condition : `pos` est un itérateur valide et `size() <= max_size()`.

post-condition : `size()` est incrémenté de 1, l'ordre des éléments est inchangé. Retourne un pointeur sur la copie de `x`.

`template <class InputIterator>`
`void insert(iterator pos,InputIterator first,InputIterator last)` *(Sequence)*

Insère une copie de l'intervalle `[first,last)` avant `pos`.

pré-condition : `[first,last)` est un intervalle valide et la somme de `size()` et de la distance de `first` à `last` est inférieure à `max_size()`.

post-condition : `size()` est incrémenté de la longueur de l'intervalle , l'ordre des éléments est inchangé.

`void insert(iterator pos,size_type n,const T& x)` *(Sequence)*

Insère `n` copies de `x` avant `pos`.

pré-condition : `pos` est un itérateur valide et `size()+n <= max_size()`.

post-condition : `size()` est incrémenté de `n`, l'ordre des éléments est inchangé. Retourne un pointeur sur la copie de `x`.

`basic_string& insert(size_type pos,const basic_string& s)` *(basic_string)*

Insère `s` avant `pos`.

Si `pos >s.size()`, lance l'exception `out_of_range`. Autrement il est équivalent à `insert(begin() + pos,s.begin(),s.end())`.

```
basic_string& insert(size_type pos,const basic_string& s,
                     size_type pos1,size_type n)
```
(basic_string)

Insère la sous-chaîne `s` avant `pos`.

Si `pos >s.size()` ou`pos1 >s.size()`, lance l'exception `out_of_range`. Sinon est équivalent à `insert(begin() + pos,s.begin() + pos1,s.begin() + pos1 + min(n,s.size() -pos1))`.

`basic_string& insert(size_type pos,const charT* s)` *(basic_string)*

Insère `s` avant `pos`.

Si `pos >s.size()`, lance l'exception `out_of_range`. Sinon est équivalent à `insert(begin() + pos,s,s + traits::length(s))`

```
basic_string& insert(size_type pos,const charT* s,
                     size_type n)
```
(basic_string)

Insère les n premiers caractères de `s` avant `pos`.

Si `pos >s.size()`, lance l'exception `out_of_range`. Sinon est équivalent à `insert(begin() + pos,s,s + n)`.

`basic_string& insert(size_type pos,size_type n,charT c)` *(basic_string)*

Insère `n` copies de `c` avant `pos`.

Si `pos >s.size()`, throws `out_of_range`. Sinon est équivalent à `insert(begin() + pos,n,c)`.

`basic_string& append(const basic_string& s)` *(basic_string)* concaténation

Ajoute `s` à `*this`.

Équivalent à `insert(end(),s.begin(),s.end())`.

`basic_string& append(const basic_string& s, size_type pos,size_type n)` *(basic_string)*

Ajoute une sous-chaîne de `s` à `*this`.

Si `pos >s.size()`, lance l'exception `out_of_range`. Sinon est équivalent à `insert(end(),s.begin() + pos,s.begin() + pos + min(n,s.size() -pos))`.

`basic_string& append(const charT* s)` *(basic_string)*

Ajoute `s` à `*this`.

Équivalent à `insert(end(),s,s + traits::length(s))`.

`basic_string& append(const charT* s,size_type n)` *(basic_string)*

Ajoute les premiers `n` caractères de `s` à `*this`.

Équivalent à `insert(end(),s,s + n)`.

`basic_string& append(size_type n,charT c)` *(basic_string)*

Ajoute `n` copies de `c` à `*this`.

Équivalent à `insert(end(),n,c)`.

`template <class InputIterator> basic_string& append(InputIterator first, InputIterator last)` *(basic_string)*

Ajoute un intervalle à `*this`.

Équivalent à `insert(end(),first,last)`.

`void push_back(charT c)` *(basic_string)*

Ajoute un seul caractère à `*this`.

Équivalent à `insert(end(),c)`

`basic_string& operator+=(const basic_string& s)` *(basic_string)*

Équivalent à `append(s)`.

`basic_string& operator+=(const charT* s)` *(basic_string)*

Équivalent à `append(s)`.

`basic_string& operator+=(charT c)` *(basic_string)*

Équivalent à `push_back(c)`.

effacement

`iterator erase(iterator p)` *(Sequence)*

Efface le caractère de position `p`.

`iterator erase(iterator first,iterator last)` *(Sequence)*

Efface l'intervalle `[first,last)`.

`basic_string& erase(size_type pos = 0,size_type n = npos)` *(basic_string)*

Efface un intervalle.

Si `pos >size()`, lance l'exception `out_of_range`. Sinon, équivalent à `erase(begin() + pos,begin() + pos + min(n,size() -pos))`.

`void clear()` *(Sequence)*

Efface le conteneur.

`void resize(size_type n,charT c = charT())` *(Sequence)*

Ajuste la taille de la chaîne à `n` caractères exactement en ajoutant ou retirant des caractères à la fin.

`basic_string& assign(const basic_string&)` *(basic_string)* affectation

Synonyme de `operator=`.

```
basic_string& assign(const basic_string& s,
                     size_type pos,size_type n)
```
(basic_string)

Affecte une sous-chaîne de `s` à `*this`.

Équivalent à (mais probablement plus rapide que) `clear()` suivi de `insert(0,s,pos,n)`.

`basic_string& assign(const charT* s,size_type n)` *(basic_string)*

Affecte les `n` premiers caractères de `s` à `*this`.

Équivalent à (mais probablement plus rapide que) `clear()` suivi de `insert(0,s,n)`.

`basic_string& assign(const charT* s)` *(basic_string)*

Affecte un tableau de caractères terminé par zéro à `*this`.

Équivalent à (mais probablement plus rapide que) `clear()` suivi de `insert(0,n)`.

`basic_string& assign(size_type n,charT c)` *(Sequence)*

Efface les caractères présents et les remplace par `n` copies de `c`.

```
template <class InputIterator>
    basic_string& assign(InputIterator first,
                         InputIterator last)
```
(Sequence)

Efface les caractères présents et les remplace par `[first,last)`.

```
basic_string& replace(size_type pos,size_type n,
                      const basic_string& s)
```
(basic_string) remplacement

Remplace une sous-chaîne de `*this` par la chaîne `s`.

Équivalent à `erase(pos,n)` suivi de `insert(pos,s)`.

```
basic_string& replace(size_type pos,size_type n,
                      const basic_string& s,
```

```
                    size_type pos1,size_type n1)
```

(basic_string)

Remplace une sous-chaîne de `*this` par une sous-chaîne de `s`.

Équivalent à `erase(pos,n)` suivi de `insert(pos,s,pos1,n1)`.

```
basic_string& replace(size_type pos,size_type n,
                    const charT* s,size_type n1)
```

(basic_string)

Remplace une sous-chaîne de `*this` avec les `n1` caractères de `s`.

Équivalent à `erase(pos,n)` suivi de `insert(pos,s,n1)`.

```
basic_string& replace(size_type pos,size_type n,
                    const charT* s)
```

(basic_string)

Remplace une sous-chaîne de `*this` par un tableau de caractères terminé par zéro.

Équivalent à `erase(pos,n)` suivi de `insert(pos,s)`.

```
basic_string& replace(size_type pos,size_type n,
                    size_type n1,charT c)
```

(basic_string)

Remplace une sous-chaîne de `*this` par `n1` copies de `c`.

Équivalent à `erase(pos,n)` suivi de `insert(pos,n1,c)`.

```
basic_string& replace(iterator first,iterator last,
                    const basic_string& s)
```

(basic_string)

Remplace une sous-chaîne de `*this` par la chaîne `s`.

Équivalent à `insert(erase(first,last),s.begin(),s.end())`.

```
basic_string& replace(iterator first,iterator last,
                    const charT* s,size_type n)
```

(basic_string)

Remplace une sous-chaîne de `*this` par les `n` premiers caractères de `s`.

Équivalent à `insert(erase(first,last),s,s + n)`.

```
basic_string& replace(iterator first,iterator last,
                    const charT* s)
```

(basic_string)

Remplace une sous-chaîne de `*this` par un tableau de caractères terminé par le caractère nul.

Équivalent à
`insert(erase(first,last),s,s + traits::length(s))`.

`basic_string& replace(iterator first,iterator last,`
`size_type n,charT c)`

(basic_string)

Remplace une sous-chaîne de `*this` par `n` copies de `c`.

Équivalent à `insert(erase(first,last),n,c)`.

`template <class InputIterator>`
`basic_string& replace(iterator first,iterator last,`
`InputIterator f,InputIterator l)`

(basic_string)

Remplace une sous-chaîne de `*this` par l' intervalle `[f,l)`.

Équivalent à `insert(erase(first,last),f,l)`.

`size_type copy(charT* buf,size_type n,size_type pos = 0) const`

(basic_string)

Copie une sous-chaîne de `n` caractères au plus de `*this` dans un tampon. Cette fonction ne termine *pas* le tampon par un caractère nul.

Lance l'exception `out_of_range` si `pos >size()`. Sinon, équivalent à `copy(begin() + pos,begin() + pos + min(n,size()),buf)`.

`size_type find(const basic_string& s,size_type pos = 0) const` recherche

(basic_string)

Recherche une sous-chaîne `s` de `*this`, en commençant au caractère `pos` de `*this`.

La seule différence avec la fonction `search` est que celle-ci utilise l'opérateur `operator==` ou un objet fonction pour la comparaison, alors que `find` utilise `traits::eq`.

`find` retourne la plus petite position `N` telle que `pos <= N`, `pos + s.size() <= size()` et pour tout `i` plus petit que `s.size()`, `(*this)[N + i]` est égal par comparaison à `s[i]`. Si il n'existe pas de telle position `N`, la fonction retourne `npos`. L'appel avec `s.size() >size() -pos` est légal mais la recherche échoue toujours.

`size_type find(const charT* s,`
`size_type pos,size_type n) const` *(basic_string)*

Recherche une sous-chaîne de `*this` formée des `n` premiers caractères `s` , en commençant au caractère `pos` de `*this`.

Équivalent à `find(basic_string(s,n),pos)`.

`size_type find(const charT* s,size_type pos = 0) const`

(basic_string)

Recherche une sous-chaîne de `*this` formée par le tableau de caractères terminé par zéro `s` , en commençant au caractère `pos` de `*this`.

Équivalent à `find(basic_string(s),pos)`.

`size_type find(charT c,size_type pos = 0) const` *(basic_string)*

Recherche le caractère `c`, en commençant au caractère `pos`.

Retourne la première position `N` plus grande ou égale à `pos`, et plus petite que `size()`, telle que `(*this)[N]` se compare comme égal à `c`. Retourne `npos` si aucune position `N` n'existe.

La seule différence avec la fonction `find_end` est que celle-ci utilise l'opérateur `operator==` ou un objet fonction pour la comparaison, alors que `find` utilise `traits::eq`.

Cette fonction membre retourne la plus grande position `N` telle que `N <= pos` et `N + s.size() <= size()` et telle que, pour tout `i` plus petit que `s.size()`, `(*this)[N + i]` se compare comme égal à `s[i]`. L'appel avec `s.size() >size()` est légal mais la recherche échoue toujours.

```
size_type rfind(const basic_string& s,
               size_type pos = npos) const
```

(basic_string)

Recherche arrière d' une sous-chaîne `s` de `*this`, en commençant au caractère `min(pos,size())` de `*this`.

```
size_type rfind(const charT* s,size_type pos,
               size_type n) const
```

(basic_string)

Recherche arrière d'une sous-chaîne de `*this` formée des `n` premiers caractères `s` , en commençant au caractère `min(pos,size())` de `*this`.

Équivalent à `rfind(basic_string(s,n),pos)`.

`size_type rfind(const charT* s,size_type pos = npos) const`

(basic_string)

Recherche arrière d'une sous-chaîne de `*this` formée par le tableau de caractères terminé par zéro `s` , en commençant au caractère `min(pos,size())` de `*this`.

Équivalent à `rfind(basic_string(s),pos)`.

`size_type rfind(charT c,size_type pos = npos) const`

(basic_string)

Recherche arrière du caractère `c`, en commençant au caractère `min(pos,size())`.

Retourne la plus grande position `N` telle que `N <= pos` et `N <size()` et telle que `(*this)[N]` se compare comme égal à `c`. Si il n'existe pas de telle position retourne `npos`

`size_type find_first_of(const basic_string& s, size_type pos = 0) const` *(basic_string)*

Recherche dans `*this`, en commençant à `pos`, du premier caractère appartenant à `s`.

La seule différence avec la fonction `find_first_of` est que celle-ci utilise l'opérateur `operator==` ou un objet fonction pour la comparaison, alors que `find` utilise `traits::eq`.

Retourne la plus petite position `N` telle que `N <= pos` et `N <size()` et telle que `(*this)[N]` se compare comme égal à un caractère de `s`. Si il n'existe pas de tel caractère retourne `npos`

`size_type find_first_of(const charT* s,size_type pos, size_type n) const` *(basic_string)*

Recherche dans `*this`, en commençant à `pos`, du premier caractère qui est aussi dans les `n` premiers de `s` c'est-à-dire dans l'intervalle `[s,s+n)`.

Retourne la plus petite position `N` telle que `N <= pos` et `N <size()` et telle que `(*this)[N]` se compare comme égal à un caractère de `[s,s+n)`. Retourne `npos` si il n'existe pas de telle position.

`size_type find_first_of(const charT* s,size_type pos = 0) const` *(basic_string)*

Recherche dans `*this`, en commençant à `pos`, du premier caractère appartenant à `s`.

Équivalent à `find_first_of(s,pos,traits::length(s))`.

`size_type find_first_of(charT c,size_type pos = 0) const` *(basic_string)*

Recherche dans `*this`, en commençant à `pos`, du premier caractère égal à `c`.

Équivalent à `find(c,pos)`.

`size_type find_first_not_of(const basic_string& s,size_type pos = 0) const` *(basic_string)*

Recherche dans `*this`, en commençant à `pos`, du premier caractère n'appartenant pas à `s`.

Retourne la plus petite position `N` telle que `N <= pos` et `N <size()` et telle que `(*this)[N]` ne se compare pas comme égal à un caractère de `[s,s+n)`. Retourne `npos` si il n'existe pas de telle position.

`size_type find_first_not_of(const charT* s,size_type pos, size_type n) const`*(basic_string)*

Recherche dans `*this`, en commençant à `pos`, du premier caractère qui n'est pas dans le `n` premiers de `s` c'est-à-dire qui n'est pas dans l'intervalle `[s,s+n)`.

Retourne la plus petite position `N` telle que `N <= pos` et `N <size()` et telle que `(*this)[N]` ne se compare pas comme égal à un caractère de `[s,s+n)`. Retourne `npos` si il n'existe pas de telle position.

`size_type find_first_not_of(const charT* s, size_type pos = 0) const`
(basic_string)

Recherche dans `*this`, en commençant à `pos`, du premier caractère n'appartenant pas à `s`.

Équivalent à `find_first_not_of(s,pos,traits::length(s))`.

`size_type find_first_not_of(charT c, size_type pos = 0) const` *(basic_string)*

Recherche dans `*this`, en commençant à `pos`, du premier caractère différent de `c`.

Retourne la plus petite position `N` telle que `N <= pos` et `N <size()` et telle que `(*this)[N]` ne se compare pas comme égal à `c`. Retourne `npos` si il n'existe pas de telle position.

`size_type find_last_of(const basic_string& s, size_type pos = npos) const` *(basic_string)*

Recherche arrière dans `*this`, en commençant à `min(pos,size())`, du premier caractère appartenant à `s`.

Retourne la plus grande position `N` telle que `N <= pos` et `N <size()` et telle que `(*this)[N]` se compare comme égal à un caractère de `s`. Retourne `npos` si il n'existe pas de telle position.

`size_type find_last_of(const charT* s,size_type pos, size_type n) const` *(basic_string)*

Recherche dans `*this`, en commençant à `min(pos,size())`, du premier caractère qui est aussi dans le `n` premiers de `s` c'est-à-dire dans l'intervalle `[s,s+n)`.

Retourne la plus grande position `N` telle que `N <= pos` et `N <size()`, et telle que `(*this)[N]` se compare comme égal à un caractère de `[s,s+n)`. S'il n'existe pas de telle position retourne `npos`.

`size_type find_last_of(const charT* s, size_type pos = npos) const` *(basic_string)*

Recherche arrière dans `*this`, en commençant à `min(pos,size())`, du premier caractère appartenant à `s`.

Équivalent à `find_last_of(s,pos,traits::length(s))`.

`size_type find_last_of(charT c,size_type pos = npos) const` *(basic_string)*

Recherche arrière dans `*this`, en commençant à `min(pos,size())`, du premier caractère égal à `c`.

Équivalent à `rfind(c,pos)`.

`size_type find_last_not_of(const basic_string& s,size_type pos = npos) const` *(basic_string)*

Recherche arrière dans `*this`, en commençant à `min(pos,size())`, du premier caractère qui n'est pas dans `s`.

Retourne la plus grande position `N` telle que `N <= pos` et `N <size()`, et telle que `(*this)[N]` ne se compare pas comme égal à un caractère de `s`. S'il n'existe pas de telle position retourne `npos`.

`size_type find_last_not_of(const charT* s, size_type pos,size_type n) const` *(basic_string)*

Recherche arrière dans `*this`, en commençant à `min(pos,size())`, du premier caractère qui n'est pas dans le `n` premiers de `s` soit dans l'intervalle `[s,s+n)`.

Retourne la plus grande position `N` telle que `N <= pos` et `N <size()` et telle que `(*this)[N]` ne se compare pas comme égal à un caractère de `[s,s+n)`. S'il n'existe pas de telle position retourne `npos`.

`size_type find_last_not_of(const charT* s,size_type pos = npos) const` *(basic_string)*

Recherche arrière dans `*this`, en commençant à `min(pos,size())`, du premier caractère n'appartenant pas à `s`.

Équivalent à `find_last_of(s,pos,traits::length(s))`.

`size_type find_last_not_of(charT c,size_type pos = npos) const` *(basic_string)* Recherche arrière dans `*this`, en commençant à `min(pos,size())`, du premier caractère différent de `c`.

Retourne la plus grande position `N` telle que `N <= pos` et `N <size()` et telle que `(*this)[N]` ne se compare pas comme égal à `c`. Retourne `npos` si il n'existe pas de telle position.

`basic_string substr(size_type pos = 0,size_type n = npos) const` *(basic_string)*

Retourne une sous-chaîne de `*this`.

Équivalent à `basic_string(*this,pos,n)`.

`int compare(const basic_string& s) const` *(basic_string)* comparaison

Comparaison lexicographique trivalente de `s` et `*this` (comme `strcmp`).

Si `traits::compare(data,s.data(),min(size(),s.size()))` est différent de zéro, retourne cette valeur ; sinon retourne un nombre négatif si

`size() <s.size()`, un nombre positif si `size() >s.size()`, et zéro si les deux sont égaux.

```
int compare(size_type pos,size_type n,
        const basic_string& s) const
```
(basic_string)

Comparaison lexicographique trivalente de `s` et une sous-chaîne de `*this`.

Équivalent à `basic_string(*this,pos,n).compare(s)`.

```
int compare(size_type pos,size_type n,const basic_string& s,
        size_type pos1,size_type n1) const
```
(basic_string)

Comparaison lexicographique trivalente d'une sous-chaîne de `s` et d'une sous-chaîne de `*this`.

Équivalent à :

`basic_string(*this,pos,n).compare(basic_string(s,pos1,n1))`.

```
int compare(const charT* s) const
```
(basic_string)

Comparaison lexicographique trivalente de `s` et `*this`.

Équivalent à `compare(basic_string(s))`.

```
int compare(size_type pos,size_type n,const charT* s,size_type
    len = npos) const
```
(basic_string)

Comparaison lexicographique trivalente des `min(len,traits::length(s))` premiers caractères de `s` et d'une sous-chaîne de `*this`.

Équivalent à :

```
basic_string(*this,pos,n).
     compare(basic_string(s,min(len,traits::length(s)))).
```

11.4 Fonctions non membres

```
template <class charT,class traits,class Alloc>
basic_string<charT,traits,Alloc>
operator+(const basic_string<charT,traits,Alloc>& s1,
          const basic_string<charT,traits,Alloc>& s2)
```
(basic_string)

Concaténation de chaînes. Fonction globale (non membre).

Équivalente à créer une copie temporaire de `s` ajouter `s2` et rendre la copie temporaire.

```
template <class charT,class traits,class Alloc>
basic_string<charT,traits,Alloc>
operator+(const charT* s1,
```

```
        const basic_string<charT,traits,Alloc>& s2)
```

(basic_string)

Concaténation de chaînes. Fonction globale (non membre).

Équivalente à créer un *(basic_string)* temporaire à partir de s1, ajouter s2 et rendre la copie temporaire.

```
 template <class charT,class traits,class Alloc>
basic_string<charT,traits,Alloc>
operator+(const basic_string<charT,traits,Alloc>& s1,
        const charT* s2)
```

(basic_string)

Concaténation de chaînes. Fonction globale (non membre).

Équivalente à créer une copie temporaire de s ajouter s2 et rendre la copie temporaire.

```
 template <class charT,class traits,class Alloc>
basic_string<charT,traits,Alloc>
operator+(charT c,
        const basic_string<charT,traits,Alloc>& s2)
```

(basic_string)

Concaténation de chaînes. Fonction globale (non membre).

Équivalente à créer un objet temporaire avec le constructeur basic-_string(1,c), ajouter s2, et rendre l'objet temporaire.

```
 template <class charT,class traits,class Alloc>
basic_string<charT,traits,Alloc>
operator+(const basic_string<charT,traits,Alloc>& s1,
        charT c)
```

(basic_string)

Concaténation de chaînes. Fonction globale (non membre).

Équivalente à créer un objet temporaire, ajouter c avec push_back, et rendre l'objet temporaire.

```
 template <class charT,class traits,class Alloc>
bool operator==(const basic_string<charT,traits,Alloc>& s1,
            const basic_string<charT,traits,Alloc>& s2)
```

(Container)

Égalité des chaînes. Fonction globale (non membre).

```
 template <class charT,class traits,class Alloc>
bool operator==(const charT* s1,
```

```
                const basic_string<charT,traits,Alloc>& s2)
```

(basic_string)

Égalité des chaînes. Fonction globale (non membre).

Équivalent à `basic_string(s1).compare(s2) == 0`.

```
 template <class charT,class traits,class Alloc>
bool operator==(const basic_string<charT,traits,Alloc>& s1,
                const charT* s2)
```

(basic_string)

Égalité des chaînes. Fonction globale (non membre).

Équivalent à `basic_string(s1).compare(s2) == 0`.

```
 template <class charT,class traits,class Alloc>
bool operator!=(const basic_string<charT,traits,Alloc>& s1,
                const basic_string<charT,traits,Alloc>& s2)
```

(Container)

Différence des chaînes. Fonction globale (non membre).

```
 template <class charT,class traits,class Alloc>
bool operator!=(const charT* s1,
                const basic_string<charT,traits,Alloc>& s2)
```

(basic_string)

Différence des chaînes. Fonction globale (non membre).

Équivalent à `basic_string(s1).compare(s2) == 0`.

```
 template <class charT,class traits,class Alloc>
bool operator!=(const basic_string<charT,traits,Alloc>& s1,
                const charT* s2)
```

(basic_string)

Différence des chaînes. Fonction globale (non membre).

Équivalent à `!(s1 == s2)`.

```
 template <class charT,class traits,class Alloc>
bool operator<(const basic_string<charT,traits,Alloc>& s1,
               const basic_string<charT,traits,Alloc>& s2)
```

(Container)

Comparaison de chaînes. Fonction globale (non membre).

```
 template <class charT,class traits,class Alloc>
bool operator<(const charT* s1,
               const basic_string<charT,traits,Alloc>& s2)
```

(basic_string)

Comparaison de chaînes. Fonction globale (non membre).

Équivalent à `!(s1 == s2)`.

```
template <class charT,class traits,class Alloc>
bool operator<(const basic_string<charT,traits,Alloc>& s1,
               const charT* s2)
```

(basic_string)

Comparaison de chaînes. Fonction globale (non membre).

Équivalent à `!(s1 == s2)`.

```
template <class charT,class traits,class Alloc>
void swap(basic_string<charT,traits,Alloc>& s1,
          basic_string<charT,traits,Alloc>& s2)
```

(Container)

Échange le contenu de deux chaînes.

```
template <class charT,class traits,class Alloc>
basic_istream<charT,traits>&
operator>>(basic_istream<charT,traits>& is,
           basic_string<charT,traits,Alloc>& s)
```

(basic_string)

Lit `s` à partir du flot d'entrée `is`.

Saute les espaces blancs et remplace le contenu de `s` par les caractères du flot d'entrée. La lecture continue jusqu'à un caractère d'espacement (qui ne sera pas extrait), ou jusqu'à une fin de fichier. Quand `is.width()` est différent de zéro la lecture s'arrête au bout de `is.width()` caractères. Cette fonction membre réinitialise `is.width()` à zéro.

```
template <class charT,class traits,class Alloc>
basic_ostream<charT,traits>&
operator<<(basic_ostream<charT,traits>& os,
           const basic_string<charT,traits,Alloc>& s)
```

(basic_string)

Écrit `s` sur le flot de sortie `os`.

Il écrit `max(s.size(),os.width())` caractères, en remplissant si nécessaire. Cette fonction membre réinitialise `os.width()` à zéro.

```
template <class charT,class traits,class Alloc>
basic_istream<charT,traits>&
getline(basic_istream<charT,traits>& is,
```

```
        basic_string<charT,traits,Alloc>& s,
        charT delim)
```

(basic_string)

Lit une chaîne dans le flot d'entrée `is` en s'arrêtant à `delim`.

Remplace le contenu de `s` par les caractères du flot d'entrée. La lecture continue jusqu'à un caractère `delim` (qui sera extrait mais pas stocké dans `s`), ou jusqu'à la fin de fichier. Au contraire de `operator«`, `getline` ne saute pas les espacements. Elle est habituellement utilisée pour lire une ligne de texte telle qu'elle apparaît dans le fichier d'entrée.

```
 template <class charT,class traits,class Alloc>
basic_istream<charT,traits>&
getline(basic_istream<charT,traits>& is,
        basic_string<charT,traits,Alloc>& s)
```

(basic_string)

Lit une ligne dans le flot d'entrée `is`.

Équivalent à `getline(is, s, is.widen('\n\))`.

Annexe A

Règles de programmation

Rec. 1.1 Utilisez des noms significatifs. (p. 20)

Rec. 1.2 Utilisez des noms anglais pour les identificateurs. (p. 20)

Rec. 1.3 Soyez cohérents pour l'attribution de noms aux fonctions, types, variables et constantes. (p. 20)

Rec. 1.4 Les seuls noms globaux doivent être les identificateurs de `namespace`. (p. 26)

Rec. 1.5 N'utilisez pas de clause ou de directive `using` globales dans un fichier en-tête. (p. 29)

Règle 1.8 Un nom ne doit pas comprendre deux barres de soulignement de suite. (p. 21)

Règle 1.9 Un nom ne doit pas commencer par une barre de soulignement. (p. 21)

Règle 2.1 Chaque fichier en-tête doit être auto-suffisant. (p. 81)

Rec. 2.4 Les définitions des fonctions en-lignes seront placées dans un fichier séparé. (p. 81)

Rec. 2.5 La définition des fonctions patrons d'une classe doit être placée dans un fichier séparé. (p. 82)

Rec. 3.1 Tout fichier doit contenir un commentaire de copyright. (p. 85)

Rec. 3.2 Tout fichier doit contenir un commentaire avec une description du contenu du fichier. (p. 85)

Rec. 3.3 Tout fichier doit contenir une chaîne constante locale qui identifie le fichier. (p. 85)

Rec. 3.4 Utilisez `//` pour les commentaires. (p. 85)

Rec. 3.5 Tous les commentaires seront écrits en anglais. (p. 85)

Règle 4.1 Ne changez pas l'index de boucle à l'intérieur d'une boucle `for`. (p. 62)

Rec. 4.2 Modifiez la variable de boucle prés de l'endroit ou se trouve la condition de sortie. (p. 62)

Rec. 4.3 Toutes les primitives de contrôle de flux (`if`, `else`, `while`, `for`, `do`, `switch`,et `case`) doivent être suivies d'un bloc éventuellement vide. (p. 58)

Rec. 4.4 Les instructions qui suivent une étiquette `case` doivent se terminer par une instruction qui sort du `switch`. (p. 60)

Rec. 4.5 Tous les `switch` doivent avoir un `default`. (p. 61)

Règle 4.6 Utilisez les `break` et `continue` à la place de `goto`. (p. 64)

Rec. 4.7 Ne créez pas de fonctions trop complexes. (p. 72)

Rec. 5.1 Déclarez et initialisez les variables près de l'endroit ou elles sont utilisées. (p. 25)

Rec. 5.2 Si possible initialisez les variables à l'endroit de leur déclaration. (p. 23)

Rec. 5.3 Ne déclarez qu'une variable par instruction de déclaration. (p. 42)

Rec. 5.4 Les littéraux ne doivent être utilisés que pour la définition des constantes et des énumérations. (p. 34)

Rec. 5.5 Initialisez toutes les données membres. (p. 136)

Règle 5.6 Ordonnez la liste des initialiseurs dans le même ordre que celui de l'entête ; d'abord les classes de bases, puis les données membres. (p. 136)

Rec. 5.8 Évitez les recopies inutiles d'objets quand la copie est coûteuse. (p. 154)

Règle 5.9 Une fonction ne doit jamais retourner un pointeur ou une référence à une variable locale en dehors de sa portée de déclaration, ni y donner accès par un autre moyen. (p. 76)

Rec. 5.10 Quand les objets d'une classe ne doivent jamais être copiés, alors le constructeur de copie et l'affectation par copie seront déclarés `private` et ne seront pas implémentés. (p. 154)

Règle 5.12 L'opérateur d'affectation doit être protégé contre la destruction d'un objet qui est affecté à lui-même. (p. 157)

Rec. 6.1 Préférez les conversions explicites aux conversions implicites. (p. 45)

Rec. 6.2 Utilisez les nouveaux opérateurs de conversion (`dynamic_cast`, `const_cast`, `reinterpret_cast`, et `static_cast`) à la place des conversions dans l'ancien style, à moins que cela ne pose des problèmes de portabilité. (p. 47)

Règle 6.3 Ne supprimez pas un `const` par une conversion. (p. 47)

Règle 6.4 Déclarez les données membres `mutable` si elles doivent être modifiées par une fonction membre constante. (p. 130)

Rec. 7.1 Déclarez `inline` les fonctions simples. (p. 132)

Rec. 7.3 Passez les arguments de types prédéfinis par valeur à moins que la fonction ne doive les modifier. (p. 74)

Rec. 7.4 N'utilisez un argument de type pointeur que si la fonction ou une fonction qu'elle appelle stocke l'adresse du pointeur. (p. 74)

Règle 7.8 Un argument paramètre ou référence doit toujours être déclaré const si la fonction ne change pas l'objet qui lui est lié. (p. 74)

Règle 7.9 Les arguments du constructeur de copie et de l'affectation par copie doivent toujours être passés par références constantes (p. 154)

Règle 7.10 Utilisez seulement des pointeurs `const char` pour accéder à des littéraux chaîne de caractères. (p. 74)

Règle 7.11 Une fonction membre qui ne change pas l'état du programme doit être déclarée `const`. (p. 129)

Règle 7.12 Une fonction qui donne un accès non `const` à la représentation d'un objet ne doit pas être déclarée const. (p. 129)

Rec. 7.13 Ne laissez pas les fonctions membres `const` changer l'état du programme. (p. 130)

Règle 7.14 Toutes les variantes des fonctions membres surchargées doivent être utilisées pour le même usage et avoir un comportement similaire. (p. 133)

Rec. 7.15 Si vous surchargez des opérations d'une famille d'opérateurs apparentés vous devez surcharger toute la famille et préserver les invariants qui existent pour les types prédéfinis. (p. 163)

Règle 7.17 Placez les arguments par défaut dans la déclaration de fonction à l'intérieur de l'en-tête, pas avec la définition de fonction dans le fichier implémentation. (p. 76)

Rec. 7.18 Les constructeurs à un argument doivent être déclarés `explicit`. (p. 158)

Rec. 7.19 N'utilisez pas les fonctions de conversion implicites. (p. 160)

Règle 8.1 `delete` doit être utilisé seulement avec `new`. (p. 49)

Règle 8.2 `delete[]` doit être utilisé seulement avec `new[]`. (p. 49)

Règle 8.3 N'accédez pas par un pointeur ou une référence à l'adresse d'un objet libéré. (p. 50)

Règle 10.1 Déclarez `private` les données membres. (p. 124)

Rec. 10.2 Si une fonction membre retourne un pointeur ou une référence, vous devez documenter la manière de l'utiliser et sa durée de validité. (p. 150)

Rec. 10.3 Les instructions de sélection `if-else` et `switch` seront utilisées quand le flux de controle dépend de la valeur de l'objet ; la liaison dynamique est utilisée quand le flux de controle dépend du type de l'objet. (p. 191)

Règle 10.4 Une classe de base publique doit avoir un destructeur virtuel ou un destructeur protégé. (p. 192)

Règle 10.5 Si l'on hérite du même parent par plusieurs classes de base, ce parent doit être une classe de base virtuelle. (p. 187)

Rec. 10.6 Spécifiez les classes en utilisant des pré-conditions, post-conditions, exceptions et invariants de classe. (p. 119)

Rec. 10.7 Utilisez **C**++ pour décrire les pré-conditions, post-conditions, et invariants de classe. (p. 119)

Règle 13.1 Utilisez `new` et `delete` à la place de `malloc calloc realloc` et `free`. (p. 49)

Règle 13.2 Utilisez la bibliothéque `iostream` à la place des entrées-sorties dans le syle du **C**. (p. 99)

Rec. 13.4 Utilisez la surcharge de fonctions et les appels chaînés plutôt que des fonctions avec un nombre inconnu d'arguments. (p. 76)

Règle 13.5 N'utilisez pas de macro-instructions à la place des constantes, `enum`, `typedef`. (p. 88)

Rec. 13.6 Utilisez une classe tableau au lieu des tableaux prédéfinis. (p. 75)

Rec. 13.7 N'utilisez pas d'unions. (p. 97)

Règle 15.4 Les en-têtes fournies par le système doivent être placées entre des crochets `<...>`, les autres en-têtes entre guillemets `"..."`. (p. 79)

Rec. 15.5 Ne mettez pas de noms absolus de répertoires dans les directives `include`. (p. 79)

Règle 15.7 Ne supposez pas que les objets ont une taille ou une disposition arbitraire en mémoire. (p. 46)

Rec. 15.8 N'utilisez pas les pragmas. (p. 96)

Rec. 15.9 Quand c'est possible utilisez un simple `int` pour stocker, passer ou retourner une valeur entière. (p. 32)

Règle 15.10 Assurez vous que vous ne tronquez pas des données significatives dans une conversion vers un type plus petit. (p. 46)

Rec. 15.11 Utilisez des `typedef` ou des classes pour cacher les types de données dépendant de l'application ou de la représentation machine. (p. 46)

Rec. 15.12 Les noms globaux (classes externes, variables, constantes `typedef`, et `enum`) doivent être préfixés quand `namespace` n'existe pas sur le compilateur utilisé. (p. 26)

Rec. 15.13 Utilisez des macro-instruction pour détourner l'utilisation des mots-clés non implémentés. (p. 88)

Règle 15.14 Ne réutilisez pas les variables déclarées dans une boucle `for`. (p. 62)

Rec. 15.15 Une seule directive `include` est nécessaire pour les `template`. (p. 83)

Rec. 15.17 Ne vous fiez pas à la durée de vie des objets temporaires. (p. 150)

Style A.2 Quand un nom est composé de plusieurs mots tous les mots sont placés de manière contiguë et commencent par une majuscule, sauf éventuellement le premier. (p. 21)

Style A.3 Les noms des classes, `typedef`, et types énumérés commencent par une majuscule. (p. 21)

Style A.4 Les noms des variables et fonctions commencent par une minuscule. (p. 21)

Style A.5 Les données membres sont postfixées par la lettre M. (p. 122)

Style A.6 Le nom des macro-instructions est entièrement en majuscules. (p. 87)

Style A.7 La garde d'un `include` doit être le nom du fichier en-tête avec tout les caractères illégaux remplacés par des caractères soulignés. (p. 81)

Style A.8 Évitez les lettres qui se confondent avec des chiffres et réciproquement. (p. 21)

Style A.9 Les fichiers en-têtes seront suffixés par `.hh` (p. 79)

Style A.10 Les fichiers de définition `inline` doivent avoir l'extension `.icc` (p. 81)

Style A.12 Donnez toujours un spécificateur d'accès pour les classes de base et les données membres. (p. 121)

Style A.13 Les sections `public`, `protected` et `private` doivent être placées dans cet ordre. (p. 121)

Style A.14 Le mot-clé `struct` sera utilisé seulement pour les structures dans le style du langage **C**. (p. 165)

Liste des Programmes

Index